GPU-Accelerated Boundary Element Method for Stress Analysis of Underground Excavations

Junjie Gu

A Thesis

in

The Department

of

Building, Civil and Environmental Engineering

Presented in Partial Fulfillment of the Requirements

for the Degree of Master of Applied Science (Civil Engineering) at

Concordia University

Montreal, Quebec, Canada

November 2015

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: Junjie Gu

Entitled: GPU-Accelerated Boundary Element Method for Stress Analysis of Underground Excavations

and submitted in partial fulfillment of the requirements for the degree of

Master of Applied Science (Civil Engineering)

complies with the regulations of the University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

Dr. A.M. Hanna	Chair, Examiner
Dr. L. Wang	Examiner
Dr. K. Skonieczny	Examiner
Dr. A.M. Zsaki	Supervisor

Approved by _____

Chair of Department or Graduate Program Director

November 2015 _____

Dean of Faculty

ABSTRACT

GPU-Accelerated Boundary Element Method for Stress Analysis of Underground Excavations

Junjie Gu

Stress analysis is one of the most important processes in designing an underground excavation. With the usage of numerical methods on a computer, such as the Boundary Element Method (BEM), the process of stress analysis can be made accurate. However, the computational implementation of stress analysis often requires considerable time and computational resources. For example, in the implementation of BEM, the finer the computational grid is, the longer time will take to compute the results. Based on the research of GPU-accelerated stress analysis in geomechanics (Zsaki, 2011), this thesis investigates one type of acceleration method, which used the parallel computing ability of modern graphics processing units (GPUs), for application to the traditional BEM algorithm. In this thesis, OpenCL was used as the framework to compile and execute programs on GPUs. By transferring and executing the most computational expensive parts of the traditional BEM code onto GPU, a respectable acceleration was achieved. Subsequently, with the application of a two-dimensional circular excavation example, the accuracy of the BEM algorithm implementation on a GPU was verified for both single-precision and double-precision calculations. In addition, two more excavation examples were taken into consideration to assess the accuracy and reduction in solution time. The performance for these three examples successfully verified the GPU-accelerating method and displayed an impressive acceleration effect with the speedup ratio of about 500 for single-precision and 15 for double-precision.

Acknowledgements

First of all, I wish to express my gratitude to my supervisor Dr. Zsaki for his helpful guidance and comments through all the steps of this research. I sincerely appreciate his continuous support for my research, which gives me the strongest conviction that I can finish my research. Without his patience and encouragement, I could not have been completed.

Thanks are also due to my dear parents: Mr. Ming Gu and Ms. Jiahong Wu, who never blamed me and always encouraged me when I encountered difficulties. It is their unconditional love and continuous support that motivate me to finish this thesis. This thesis is dedicated to them.

I would like to take this opportunity to thank all my lovely families in Vancouver and back in China for their understanding and support for my study and for their caring of my parents. Special thanks are due to my respected aunt Ms. Jinmei Xu for her love wherever she was.

Furthermore, I would like to recognize and thank my close friend Mr. Chao Li for his continuous caring and encouragement; Mr. Pengfei Zhao for his technical support and encouragement; Mr. Javad Mirvalad for his caring and technical support. I would also thank all my friends who helped me in Canada and back home for their encouragement and valuable contributions.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

b_i	Body forces (MPa)
C	Boundary of a two-dimensional region R
C_i	i-th boundary element
C_j	j-th boundary element
E	Young's modulus (MPa)
G	Shear modulus(=μ) (MPa)
l_i	Length of the element C_i
n	Outward normal vector
n̂	Unit outward normal vector
p^*	Fundamental solution
p	Traction vector
p_i	Tractions (MPa)
r	Radial distance (m)
R	A two-dimensional region
u	Displacement (m)
u^*	Fundamental solution
u	Displacement vector
w_i	Weights in logarithmic Gauss quadrature
W_i	Weights in Gauss quadrature
δ_{ij}	Kronecker delta
ε_{ii}	Normal strain
ε_{ij}	Shearing strain
λ, μ	Lame's constants
ν	Poisson's ratio
σ_{ii}	Normal stresses (MPa)
σ_{ij}	Shearing stresses (MPa)
τ_{xy}	Shearing stresses (MPa)

1 Introduction

For the majority of geotechnical projects, such as underground excavations in general and tunnels excavated in a rock mass in particular, the most important design element is to determine the stress distribution around them. Some stresses exist in the rock mass prior to excavation (e.g. in situ stresses), but the stability of an excavation is mainly governed by excavation-induced stresses during and after construction. Alongside stresses, the displacements, which will be induced by the process of excavation in the stressed rock are important for the stability as well. The method of determining the stress and displacement distributions is called stress analysis. Since the material in the underground excavation or tunnelling is rock, not all the stress analysis methods can be used. There are number of stress analysis methods that can be used in geomechanics. The boundary element method (BEM) is one among all the stress analysis methods that has considerably evolved in the past few decades. Some notable contributions to stress analysis using the BEM in geomechanics include determining stresses and displacements around long openings in a triaxial stress field with BEM (Brady and Bray, 1978), the computation of two-dimensional stress intensity factor (Blandford et al., 1981) and the introduction of elastostatic deformation field analysis using 3D mixed boundary elements (Cayol and Cornet, 1997).

With the usage of BEM algorithm on a computer, the process of stress analysis can be made accurate and efficient. For the computational implementation of stress analysis using BEM, the computational grid size for field quantities (such as stresses and displacements within the rock mass) should be as fine as possible, so that the solution results will be as accurate and detailed as they can be. However, as the computational grid size is increased, the computing time grows as well for a traditional BEM program. For example, the processing time for a small and simple circular excavation in rock, which was executed on a typical processor as of this writing (a dual core Intel CPU i3-3220) with the grid size of 500^2 was up to 16 seconds; when the grid size was increased to 1000^2, the corresponding computing time reached 62 seconds; even further, when the grid size was further increased to 1500^2, the corresponding computing time was as long as 136 seconds. It is not hard to guess how many hours the program will take if the grid size is increased up to tens or hundreds of thousands for a real project simulation. Therefore, to address the issue

of lengthy computation time, an acceleration method to the traditional BEM program is developed in this thesis.

The proposed acceleration method is to implement the BEM algorithm running in parallel on a Graphics Processing Unit (GPU). By transferring the computationally intensively calculations from the serial device (Central Processing Unit, CPU) to the parallel device (GPU) using the OpenCL framework, a considerable acceleration can be achieved. Taking the same circular excavation as example, the execution of the parallel program on a NVIDIA GTX 650Ti GPU took only 0.0292 seconds for the grid size of 500^2; the processing time for the grid size of 1000^2 was 0.1073 seconds; and finally, the processing time for the grid size of 1500^2 was as fast as 0.2378 seconds. From the test results, it was calculated that a speedup of the acceleration program over the traditional program was up to 580 for a circular excavation example. In addition, with the usage of the proposed acceleration method, we are able to analyze even larger and more complicated excavation problems with faster computers. Speedingup the solution time is always a present trend. Moreover, it is human nature to do accelerations on multiple problems, for the reason that people always wants to solve larger problems, no matter how fast our computers get.

1.1 Thesis Objectives

The objective of this thesis is to perform stress analysis using a BEM algorithm in a faster, more efficient way, through parallel computing on a GPU. To achieve this, the basic concepts of BEM will be introduced, in order to explain the implementation of the BEM algorithm with the traditional serial device CPU. From the implementation of the BEM algorithm, an analysis will be carried out to determine what will be the most computationally expensive parts of the BEM algorithm, so that acceleration can be performed on those specific parts. To develop the accelerated algorithm, concepts of parallel computing and the background of OpenCL will be introduced, so that the implementation of the BEM algorithm with the accelerated parallel device GPU can be expressed. Subsequently, the most important part is to verify the result and to analyze the acceleration performance, which is considerably valued in the development of any new algorithm and computational technique.

1.2 Thesis Outline

As an introduction, Chapter 2 gives general background information on several common numerical stress analysis methods used in geomechanics, such as the finite difference method (FDM), the finite element method (FEM), the discrete element method (DEM) and the boundary element method (BEM). Subsequently, a detailed description of the BEM will be given, including its mathematical model, integral equations, and solution techniques. In addition, a discussion about CPU and GPU computing will be carried out for the clarification of some basic concepts like serial computing, parallel computing and GPGPU. An introduction to OpenCL will also be presented together with some other environments for programming GPUs.

Chapter 3 will be devoted to the implementation of a BEM algorithm on a CPU and the acceleration implementation of the BEM algorithm on a GPU using OpenCL functions. The salient equations and the corresponding code segments will be presented to explain the programming implementation of the new, accelerated BEM algorithm. In addition, the usage of OpenCL will be presented in the parallel implementation of the BEM on a GPU.

Chapter 4 introduces an example of a circular excavation, which will be used to verify the solution accuracy of the serial program and subsequently, the parallel program. From the performance achieved for the circular excavation case, comparisons of speedup will be made between the serial program and the parallel program. A discussion on further fine-tuning and optimizing acceleration performance of the parallel program will be carried out as well.

Chapter 5 will present two practical tunnel excavation problems for GPU acceleration application: a horse-shoe shaped tunnel and a hydro-electric turbine power cavern. The results of the serial program and parallel program will be compared and the acceleration effect will be shown with the speedup ratio diagrams for these two cases.

Finally, Chapter 6 summarizes the work done and concludes this thesis and gives recommendations for further study.

2 Literature Review -- Numerical Stress Analysis in Geomechanics and GPU Acceleration of Numerical Computation Using OpenCL

Stress analysis in geomechanics acts an important role in designing a tunneling or an underground excavation. In the project of a tunneling or an excavation, there are generally two processes: design process and construction process. In the reference of a widely applied tunneling method, the New Austrian Tunneling Method (NATM), the design process follows three steps (Brown, 1981): first, with the controlled deformation of the rock mass around an excavation, calculate the strength in the rock mass and mobilize the strength to the maximum value; second, design initial support for safe tunneling construction; third, monitor the deformation of the initial support system for the safety of construction process. In the process of construction, NATM also gives practical information such as building the initial support system with combination of shortcrete and reinforcements (fiber, welded wire-mesh and steel arches), building the permanent support with concrete lining.

The importance of stress analysis mainly lies in the design process of an excavation, where several requirements must be met in the process of design (Hazegh and Zsaki, 2013): the local and overall stability of an excavation must be computed and the initial support system must be guaranteed; the induced displacements of an excavation must be designed within a controlled value and the effect to its nearby constructions must be under control.

Because there were several unfortunate collapses in the past and was possible due to the error in design process (underestimating the stresses and displacements in critical regions of problem domains) and other unexpected ground conditions, that's why we are doing the optimization research of stress analysis in geomechanics.

2.1 Numerical Stress Analysis in Geomechanics

Quite a few important problems in science and engineering can be classified as boundary value problems. The mathematical definition of these boundary value problems consists of a partial differential equation and certain constraints or conditions that need to be met. The partial differential equation models the physics of the problem in a region of interest R, and the boundary conditions are constraints specified on the boundary C. There are two types of

problems defined based on the boundary element technique: interior problems and exterior problems, as Figure 2.1a and Figure 2.1b shows. The difference between Figure 2.1a and Figure 2.1b is that Figure 2.1a displays a region R bounded by an outside contour C, which represents interior problems; while in Figure 2.1b, the infinite plane R and its inside auxiliary contour C' represent exterior problems.

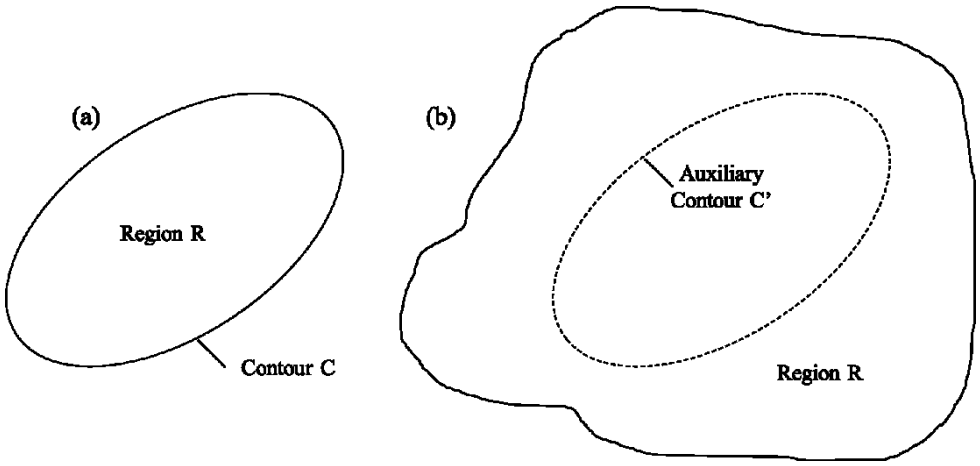


Figure 2.1: The contour C, contour C' and the region R in boundary value problems.

Generally, there are two ways to solve boundary value problems: analytical solutions and numerical solutions. Analytical solutions to boundary value problems can be easily derived when the region R is homogeneous, or when the geometry is simple, or when the boundary conditions are relatively straightforward and when the governing partial differential equations are linear (Crouch et al., 1983). However, analytical solutions cannot be found for most 'real world' problems. For example, geomaterials such as soils and rocks display non-linear behavior, or the problem domain may not be homogeneous or isotropic, or the geometry of the problem may be complex. Thus an approximate solution must be found using a numerical method on a computer.

With the advancement of computer science and numerical computation, as applied to multiple fields, several numerical methods have attracted the interest of geomechanics researchers. These numerical methods are: the finite difference method (FDM), the finite element method (FEM), the infinite element method (IEM), the boundary element method (BEM), the discrete element method (DEM), and the discrete fracture network method (DFN). These methods can be classified into three types:

1) Continuum methods, including FDM, FEM, IEM and BEM;

2) Discontinuum methods, including DEM, DFN;

3) Hybrid continuum/discrete methods.

The main theme of this thesis is the application of the BEM, therefore, a detailed explanation will be presented regarding to foundations and formulation of BEM. While, for better understanding of the BEM in the context of numerical stress analysis, a brief introduction of other most used numerical methods will be first presented in the following sections.

2.1.1 Finite Difference Method (FDM)

The FDM is the oldest numerical method applied to computer simulation (Peiró and Sherwin, 2005). The fundamental feature of this method is the discretization of a solution domain into a difference grid, and replacing the continuum with the finite points on the grid. At the same time, the partial derivatives in the partial differential equations (PDE) are replaced by differences defined at grid points. In other words, this method approximates the solutions of PDEs with the values of a combination of linear functions.

Based on the accuracy of the finite difference approximations, the method can be divided into several forms: first-order difference approximations, second-order difference approximations, and higher-order approximations. Based on the formation of the Taylor-series expansion, which is given in Equation (2.1), the method can be organized into three types: forward difference, backward difference and central difference.

$$u(x) = \sum_{n=0}^{\infty} \frac{(x-x_i)^n}{n!} \left(\frac{\partial^n u}{\partial x^n}\right)_i, \quad u \in C^{\infty}([0,X]) \tag{2.1}$$

$$\left(\frac{\partial u}{\partial x}\right)_i \approx \frac{u_{i+1}-u_i}{\Delta x} \quad \text{forward difference}$$

$$\left(\frac{\partial u}{\partial x}\right)_i \approx \frac{u_i-u_{i-1}}{\Delta x} \quad \text{backward difference} \tag{2.2}$$

$$\left(\frac{\partial u}{\partial x}\right)_i \approx \frac{u_{i+1}-u_{i-1}}{2\Delta x} \quad \text{central difference}$$

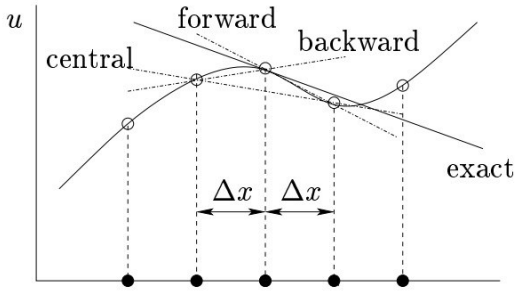


Figure 2.2: Geometric interpretation of first-order difference approximations.

As an example, Equation (2.2) and Figure 2.2 show the first-order difference approximations of Taylor series expansion.

In comparison with FEM and BEM, the FDM has several advantages (Jing and Hudson, 2002):

- The formation and solution of the equations are localized instead of global matrix system, thus handling memory and storage for computer operation becomes more efficient.
- The solution of the PDEs is more direct and intuitive because there are no local trial functions to approximate the PDE.
- Provides more straightforward simulation of complex constitutive material behaviour, such as plasticity and damage.

More often than not, the regular grid systems cannot deal with fractures, complex boundary conditions and material heterogeneity, which makes the standard FDM unsuitable for modelling practical rock mechanics problems.

2.1.2 Finite Element Method (FEM)

After the FDM, the FEM appeared in the late 1960s and early 1970s, when the regular grids of traditional FDM could not satisfy essential requirements for continuum geomechanics problems (Jing and Hudson, 2002). Based on variational principles and weighted residual methods, the main technique of the FEM is to discretize a solution domain into finite arbitrary

units and by meeting the boundary value requirements, to solve approximate solutions of each unit.

The following steps present a basic outline of the FEM method (Potts and Zdravkovic, 1999):

1) Element discretization: the geometry of a boundary value problem is replaced by an equivalent finite element mesh, which is composed of small regions called finite elements. Key points on the element are called nodes. They can be defined on the element boundaries, or within the element as Figure 2.3 shows.
2) Primary variable approximation: based on the node numbers and the variation rules, a primary variable (e.g. displacements, stresses etc.) is selected. Since the nodal values perform the variation, and the geometry of each element is regular, the rules of variation over a finite element can be established.
3) Element equations: element equations can be derived by using an appropriate variational principle (e.g. Minimum potential energy). The standard element equation is as follows (for static equilibrium) :

$$[K_E]\{\Delta d_E\} = \{\Delta R_E\} \tag{2.3}$$

where $[K_E]$ is the element stiffness matrix, $\{\Delta d_E\}$ is the vector of incremental element nodal displacements and $\{\Delta R_E\}$ is the vector of incremental element nodal forces.

4) Global equations: this can be derived by combining element equations:

$$[K_G]\{\Delta d_G\} = \{\Delta R_G\} \tag{2.4}$$

where $[K_G]$ is the global stiffness matrix, $\{\Delta d_G\}$ is the vector of all incremental nodal displacements and $\{\Delta R_G\}$ is the vector of all incremental nodal forces.

5) Boundary conditions: load and displacement conditions form the boundary conditions, which will modify the global equations. Loading conditions (e.g. line and point loads, pressures and body forces) affect $\{\Delta R_G\}$, and the displacement conditions affect $\{\Delta d_G\}$.

6) Solve the global equations: once the global stiffness matrix has been established and the boundary conditions added, it mathematically forms a large system of simultaneous equations. With appropriate numerical method (direct or iterative solvers), nodal values can be obtained by solving the global equations.

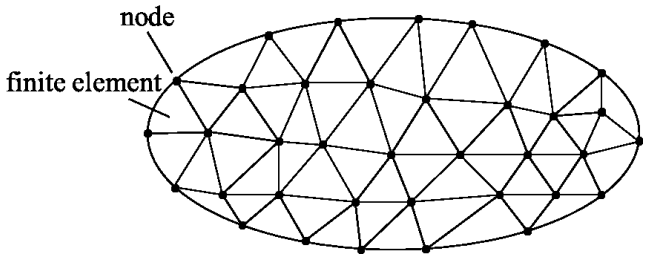


Figure 2.3: Elements and nodes in FEM.

Since its initial development, the FEM is perhaps the most popular numerical method in the fields of science and engineering (Jing and Hudson, 2002). The advantage of FEM is the flexibility in dealing with complex boundary conditions, material heterogeneity, and non-linear deformability. However, the fundamental assumption of this method is the continuum assumption, which means the FEM has a lot of limitations in treating large-scale deformation, sliding, and complete detachment of elements.

2.1.3 Discrete Element Method (DEM)

The DEM was proposed by Cundall and Strack (Cundall and Strack, 1979), and was originally applied to geotechnical mechanics. In contrast to FEM, which is limited to continuum problems, the DEM is dedicated to granular and discontinuous material problems. The main concept of this method is to discretize an object into a number of small, rigid particles and to compute the motion and effect of these particles. Based on Newton's law of motion, the method builds the equations of motion for each particle, therefore the solutions of each particle equation leads to the derivation of the whole deformation and displacement field. The equations of Newton's laws of motion are given as below:

$$m_i \frac{dv_i}{dt} = \sum_{j=1}^{N} \left(f_{c,ij} + f_{d,ij} \right) + m_i g \tag{2.5}$$

$$I_i \frac{d\omega_i}{dt} = \sum_{j=1}^{N} T_{ij} \tag{2.6}$$

where m_i and v_i are the mass and velocity of particle i, N is the number of particles in contact with current particle, $f_{c,ij}$ and $f_{d,ij}$ are the contact and viscous contact damping forces, I_i is the moment of inertia of particle i , ω_i is its angular velocity, and T_{ij} is the torque arising from contact forces, which will cause the particle to rotate (Lim, 2008).

In recent years, the DEM became one of the most rapidly developing areas of computational mechanics (Jing and Hudson, 2002). In particular, with the advent of powerful computing facilities in many parts in the world, it is possible to perform accurate and complex computer simulations involving millions of particles. Due to the explicit representation of fracture, the DEM has allowed the exploration of many applications in rock engineering (Jing and Hudson, 2002). This method can also be used to simulate a wide variety of granular flow and solid-fluid multi-phase systems (Lim, 2008). The DEM is superior to continuum-based methods, while the computational power limits the maximum number of particles and the duration of a virtual simulation (Kaixin and Lingtian, 2003).

2.1.4 Boundary Element Method (BEM)

The BEM is another continuum-based numerical method similar to FDM and FEM. The primary basis of the BEM is as follows: for a definition of a domain, only the boundary is divided into elements, and a combination of solutions will be chosen on these discrete boundary elements to satisfy the boundary conditions.

Compared to other two widely applied numerical methods, FEM and FDM, the BEM has its advantages (Jing and Hudson, 2002):

- Reduction of model dimension by one.
- Smaller system of algebraic equation.
- Simpler input data preparation.
- Solutions inside the domain are continuous.

However, the BEM shows the weakness in dealing with heterogeneous or non-linear problems (Kythe, 1995).

A representative BEM application is performed as follows: start with building a mathematical model, which consists of partial differential equations (PDE), then using the fundamental solutions of BEM to transfer PDEs into integral equations. After building the boundary integral equations from the integral equations, discretization of the boundary will be applied, and values on the boundary will be approximated by the values of the nodes and elements of each boundary element. Afterwards, solutions on the boundary can derive the solutions of interior/exterior points. A typical application of two-dimensional BEM linear elasticity problem is introduced in the following part of this section.

2.1.4.1 Mathematical Model

The linear elastic boundary value problems are characterized by a region R bounded by a contour C. Region R can either be finite or infinite. In order to analyze the stresses and displacements for an arbitrary point within the region R, an infinitesimal element, as Figure 2.4 shows, is taken into consideration. In a Cartesian coordinate system, where the coordinates are denoted by $\mathrm{x} = (x_1, x_2, x_3)$, the equilibrium equations of stress are as follows:

$$\sigma_{ij} = \sigma_{ji}, \quad i,j = 1,2,3, \tag{2.7}$$

$$\frac{\partial \sigma_{ij}}{\partial \mathrm{x}_i} + b_i = 0, \quad i,j = 1,2,3, \tag{2.8}$$

where b_i are the body forces. Corresponding to these stresses, the normal and shearing strains are defined as follows:

Normal strains: $\varepsilon_{ii} = u_{i,i}, \quad i = 1,2,3,$

Shearing strains: $\varepsilon_{ij} = \frac{1}{2}\left(u_{i,j} + u_{j,i}\right), \quad i,j = 1,2,3, \quad (i \neq j)$ (2.9)

where (u_1, u_2, u_3) are translations along the (x_1, x_2, x_3) directions, respectively.

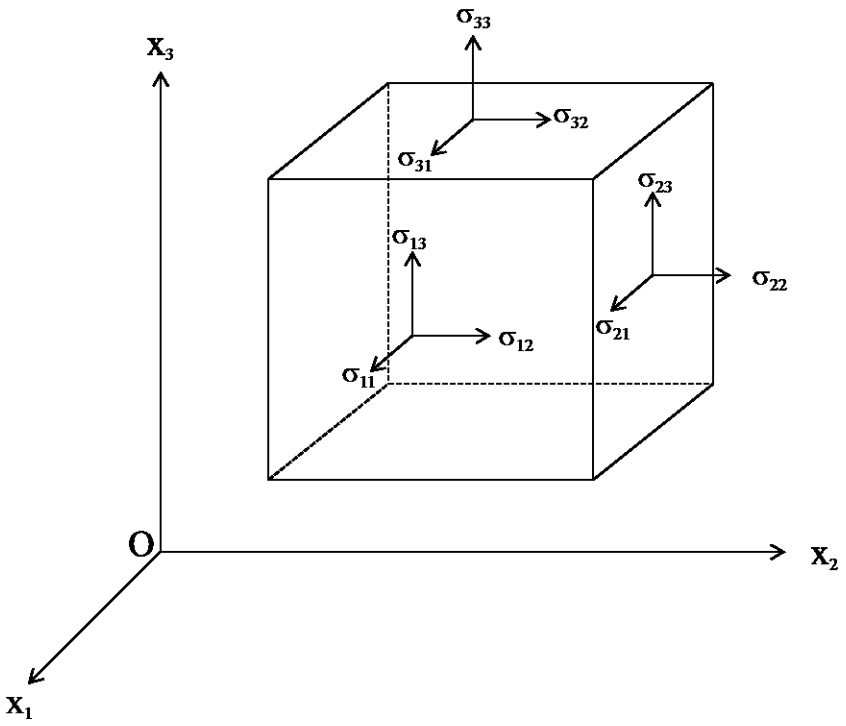


Figure 2.4: Stresses on an infinitesimal element.

The strain-stress relations for an isotropic material are given by the generalized Hooke's law:

$$\varepsilon_{11} = \frac{\sigma_{11} - v\sigma_{22} - v\sigma_{33}}{E}, \quad \varepsilon_{12} = \frac{\sigma_{12}}{G},$$

$$\varepsilon_{22} = \frac{-v\sigma_{11} + \sigma_{22} - v\sigma_{33}}{E}, \quad \varepsilon_{23} = \frac{\sigma_{23}}{G}, \tag{2.10}$$

$$\varepsilon_{33} = \frac{-v\sigma_{11} - v\sigma_{22} + \sigma_{33}}{E}, \quad \varepsilon_{31} = \frac{\sigma_{31}}{G},$$

where E is the Young's modulus, G is the shear modulus, v is the Poisson's ratio ($0< v<1/2$). The equation relating G, E, v is as follows:

$$\mu = \frac{E}{2(1+v)} = G \ , \lambda = \frac{E\varepsilon}{(1+v)(1-2v)} \tag{2.11}$$

where λ, μ are the Lame's constants.

There are three types of boundary conditions for linear elastic problems (Kythe, 1995):

1) Essential conditions (Dirichlet-type): the displacement u is prescribed on the boundary C of a

region R, i.e.,

$$\boldsymbol{u}(\boldsymbol{x}) = \boldsymbol{u}^0(\boldsymbol{x}), \quad \mathbf{x} \in \mathrm{R}, \tag{2.12}$$

2) Natural conditions (Neumann-type): the stresses are prescribed on the boundary C, i.e.,

$$\boldsymbol{p}(\boldsymbol{x}) = \boldsymbol{p}^0(\boldsymbol{x}), \quad \mathbf{x} \in \mathrm{C}, \tag{2.13}$$

where **p** is the traction vector at a point x ∈ C, and n̂ is the unit outward normal at that point as Figure 2.5 shows. Since $\mathrm{p} = \sigma \cdot \hat{\mathrm{n}}$, the following three equilibrium conditions are satisfied on the boundary:

$$\left.\begin{aligned} \sigma_{11}n_1 + \sigma_{21}n_2 + \sigma_{31}n_3 &= p_1^0 \\ \sigma_{12}n_1 + \sigma_{22}n_2 + \sigma_{32}n_3 &= p_2^0 \\ \sigma_{13}n_1 + \sigma_{23}n_2 + \sigma_{33}n_3 &= p_3^0 \end{aligned}\right\}, \tag{2.14}$$

where $n_i = \cos(n, x_i)$. Equation (2.14) can also be written as

$$p_i = \sigma_{ij}n_j = p_i^0, \quad i,j = 1,2,3. \tag{2.15}$$

Thus, an equilibrium stress field is defined on a set of sufficiently continuous stress functions σ_{ij}, which satisfy Equation (2.8), (2.15) and the Hooke's law

$$\sigma_{ij} = \lambda\varepsilon_{kk}\delta_{ij} + 2\mu\varepsilon_{ij}, \quad i,j,k = 1,2,3. \tag{2.16}$$

Substituting Equation (2.9) into (2.16), then substituting the equation into Equation (2.15), the natural conditions in terms of the displacement **u** can be written as

$$\lambda\hat{\mathrm{n}}\nabla \cdot u + 2\mu\frac{\partial \boldsymbol{u}}{\partial n} + \mu(\hat{\mathrm{n}} \times curl\ \boldsymbol{u}) = \boldsymbol{p}^0(\boldsymbol{x}). \tag{2.17}$$

3) Mixed conditions: a combination of the first two types which applied on different disjoint portions of the boundary where displacements are prescribed on one portion and stresses on the other. Another kind of mixed conditions is the Robin-type and is shown as follows:

$$a\mathbf{u} + b\mathbf{p} = c, \tag{2.18}$$

where *a*,*b*,*c* are constants.

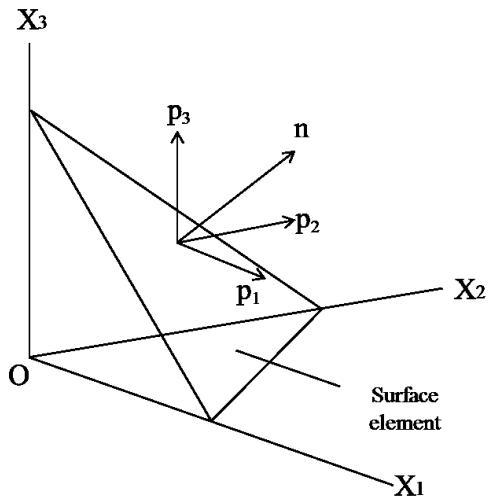


Figure 2.5: Surface tractions. (Kythe, 1995)

2.1.4.2 Integral Equations and Fundamental Solutions

The two-dimensional linear elastic boundary value problems are classified into two types: plane stress problems and plane strain problems (Crouch et al., 1983). Plane stress means that the stresses are restricted to a single plane, in which case $\sigma_{13} = \sigma_{23} = \sigma_{33} = 0$. Plane strain means that the strains are restricted to a single plane, in which case $\varepsilon_{13} = \varepsilon_{23} = \varepsilon_{33} = 0$. In both cases, there are eight components: stresses σ_{11}, σ_{22}, σ_{12}, strains ε_{11}, ε_{22}, ε_{12}, and displacements u_1, u_2.

The formulation of two-dimensional elasticity problems is as follows:

1) Equilibrium equations:

$$\sigma_{ij,i} + b_i = 0, \;\; i,j = 1,2. \tag{2.19}$$

Boundary conditions:

$$p_i = \sigma_{ij} n_j = p_i^0, \;\; i,j = 1,2. \tag{2.20}$$

2) Geometry equations:

$$\varepsilon_{ij} = \frac{1}{2}\left(u_{i,j} + u_{j,i}\right), \;\; i,j = 1,2. \tag{2.21}$$

3) Stress-strain relations can be written in the form:

For a plane-strain case:

$$\varepsilon_{ij} = \frac{1}{2G}\left[\sigma_{ij} - v\sigma_{kk}\delta_{ij}\right], \;\; i,j,k = 1,2. \tag{2.22}$$

and

$$\sigma_{ij} = 2G\left[\varepsilon_{ij} + \frac{v}{1-2v}\varepsilon_{kk}\delta_{ij}\right], \;\; i,j,k = 1,2. \tag{2.23}$$

For a plane-stress case:

$$\varepsilon_{ij} = \frac{1}{2G}\left[\sigma_{ij} - \frac{v}{1+v}\sigma_{kk}\delta_{ij}\right], \;\; i,j,k = 1,2. \tag{2.24}$$

and

$$\sigma_{ij} = 2G\left[\varepsilon_{ij} + \frac{v}{1-v}\varepsilon_{kk}\delta_{ij}\right], \;\; i,j,k = 1,2. \tag{2.25}$$

where $\delta_{ij} = \begin{cases} 0, \;\; if \; i \neq j \\ 1, \;\; if \; i = j \end{cases}$ is called the Kronecker delta.

Substituting Equation (2.21) into (2.23), then combining with Equation (2.19), the equilibrium equation in terms of displacements for an isotropic plane strain case can be written as:

$$\lambda u_{k,ki} + G\left(u_{i,jj} + u_{j,ij}\right) + b_i = 0 \;\; (i,j,k = 1,2). \tag{2.26}$$

In order to solve the elastic boundary value problem, the fundamental solution $u^*_{lk}(P,Q)$ should be derived first. Consider a homogeneous isotropic elastic solid with a region R and a boundary C. P and Q are two points in the region R. From Equation (2.26), the fundamental solution for the displacement satisfies the following equation:

$$\lambda u^*_{lk,ki}(P,Q) + G\left[u^*_{li,jj}(P,Q) + u^*_{lj,ij}(P,Q)\right] + \delta(P,Q)\delta_{li} = 0 \;\; (i,j,k,l = 1,2). \tag{2.27}$$

The fundamental solution $u^*_{lk}(P,Q)$ represents the displacements at point Q due to a concentrated unit load at point P in the direction *l*. From solving Equation (2.27), the fundamental solutions for an isotropic plane strain case are:

$$\left.\begin{aligned} u^*_{lk}(P,Q) &= \frac{1}{8\pi\mu(1-v)}\left[(3-4v)\ln\left(\frac{1}{r}\right)\delta_{lk} + \frac{\partial r}{\partial x_l}\frac{\partial r}{\partial x_k}\right] \\ p^*_{lk}(P,Q) &= -\frac{1}{4\pi(1-v)r}\left[\frac{\partial r}{\partial n}\left\{(1-2v)\delta_{lk} + 2\frac{\partial r}{\partial x_l}\frac{\partial r}{\partial x_k}\right\}\right. \\ &\quad \left. -(1-2v)\left(\frac{\partial r}{\partial x_l}n_k - \frac{\partial r}{\partial x_k}n_l\right)\right] \end{aligned}\right\}. \tag{2.28}$$

For an isotropic plane stress case, the displacements and the tractions are:

$$\left.\begin{aligned} u_{lk}^{*}(P,Q) &= \frac{1+v}{4\pi E}\left[(3-v)\ln\left(\frac{1}{r}\right)\delta_{lk} + (1+v)\frac{\partial r}{\partial x_l}\frac{\partial r}{\partial x_k}\right] \\ p_{lk}^{*}(P,Q) &= -\frac{1}{4\pi r}\left[\frac{\partial r}{\partial n}\left\{(1-v)\delta_{lk} + 2(1+v)\frac{\partial r}{\partial x_l}\frac{\partial r}{\partial x_k}\right\}\right. \\ &\quad \left. -(1-v)\left(\frac{\partial r}{\partial x_l}n_k - \frac{\partial r}{\partial x_k}n_l\right)\right] \end{aligned}\right\}. \tag{2.29}$$

While in a three-dimensional medium, the fundamental solutions for an isotropic body are:

$$\left.\begin{aligned} u_{lk}^{*}(P,Q) &= \frac{1}{16\pi\mu(1-v)r}\left[(3-4v)\delta_{lk} + \frac{\partial r}{\partial x_l}\frac{\partial r}{\partial x_k}\right] \\ p_{lk}^{*}(P,Q) &= -\frac{1}{8\pi(1-v)r^2}\left[\frac{\partial r}{\partial n}\left\{(1-2v)\delta_{lk} + 3\frac{\partial r}{\partial x_l}\frac{\partial r}{\partial x_k}\right\}\right. \\ &\quad \left. -(1-2v)\left(\frac{\partial r}{\partial x_l}n_k - \frac{\partial r}{\partial x_k}n_l\right)\right] \end{aligned}\right\}. \tag{2.30}$$

where r is the distance from point P and point Q; n_j are the direction cosines, and δ_{ij} is the Kronecker delta. Note that for a spherical coordinate system, defined by

$$\left.\begin{aligned} x_1 &= r\sin\theta\cos\phi\,, \\ x_2 &= r\cos\theta\sin\phi\,, \\ x_3 &= r\cos\theta\,, \end{aligned}\right\} \qquad 0 \le \theta \le 2\pi, 0 \le \phi \le \pi, \tag{2.31}$$

where $\frac{\partial r}{\partial x_k} = \frac{r}{x_k}$, and

$$\frac{\partial r}{\partial x_l}n_k - \frac{\partial r}{\partial x_k}n_l = \frac{\partial r}{\partial x_l}\frac{\partial x_k}{\partial r} - \frac{\partial r}{\partial x_k}\frac{\partial x_l}{\partial r} = 0. \tag{2.32}$$

For a given two-dimensional elasticity boundary value problem:

$$\lambda u_{k,ki} + G\left(u_{i,jj} + u_{j,ij}\right) = 0 \text{ in region R,} \tag{2.33}$$

$$u_i = u_i^0 \text{ on boundary } C_1,$$

$$p_i = \sigma_{ij}n_j = \lambda u_{k,k}n_i + G(u_{i,j}n_j + u_{j,i}n_j) = p_i^0 \text{ on boundary } C_2, \tag{2.34}$$

where $\mathrm{C} = \mathrm{C}_1 \cup \mathrm{C}_2$ is the boundary of a region R. P and Q are two points in the region R, P' and Q' are two points on the boundary C. Multiplying the fundamental solution u_{lk}^{*} with both sides of Equation (2.33), and performing integration within the region R, Equation (2.33) becomes:

$$\int_R \left[\lambda u_{k,ki} + G\left(u_{i,jj} + u_{j,ij}\right)\right] u_{lk}^* \, dR = 0 \tag{2.35}$$

Using Green's theorems, the PDEs in the domain can be represented by PDEs on the boundary:

$$\int_R u_{i,i} \, dR = \int_C u_i n_i \, dC\,, \quad i = 1,2 \tag{2.36}$$

After calculation, the integral relation between the displacements of point P $u_l(P)$, tractions of point Q' $p_k(Q')$ and displacements of point Q' $u_k(Q')$ yields:

$$u_l(P) = \int_C [u_{lk}^*(P, Q') p_k(Q') - u_k(Q') \cdot p_{lk}^*(P, Q')] \, dC(Q') \tag{2.37}$$

2.1.4.3 Boundary Integral Equations

Using Equation (2.37), displacements can be calculated for any point in a region. However, in order to solve the unknowns on the boundary, it's necessary to formulate boundary integral equations. By choosing a point P in the region, when this point is on the boundary, the fundamental solution has a logarithmic singularity. Thus, considering a semicircle of radius ε surrounding the boundary point Q', and Q' is at the center of this semicircle. While θ being the internal angle (in radians) at the corner at node Q'. Further, assuming that the boundary C of the region R is smooth, and that $C = C_\varepsilon + (C - C_\varepsilon)$. Equation (2.37) can be written as

$$u_l(P') = \int_{C - C_\varepsilon} [u_{lk}^*(P', Q') p_k(Q') - u_k(Q') \cdot p_{lk}^*(P', Q')] \, dC(Q')$$

$$+ \int_{C_\varepsilon} [u_{lk}^*(P', Q') p_k(Q') - u_k(Q') \cdot p_{lk}^*(P', Q')] \, dC(Q') \tag{2.38}$$

As ε → 0, the semicircle will reduce to the boundary point Q'. Taking the limit of the first integral in Equation (2.38) yields:

$$\lim_{\varepsilon \to 0} \int_{C - C_\varepsilon} [u_{lk}^*(P', Q') p_k(Q') - u_k(Q') \cdot p_{lk}^*(P', Q')] \, dC(Q')$$

$$= \int_C [u_{lk}^*(P', Q') p_k(Q') - u_k(Q') \cdot p_{lk}^*(P', Q')] \, dC(Q') \tag{2.39}$$

Considering the second integral in Equation (2.38), when $l = 1$, substituting the fundamental solution into the first part of second integral yields

$$\lim_{\varepsilon\to 0}\int_{C_\varepsilon} u_{lk}^{*}(P',Q')p_k(Q')\,dC(Q')$$

$$=\lim_{\varepsilon\to 0}\frac{1}{8\pi G(1-v)}\int_0^{\alpha}\left[p_1(3-4v)\,ln\frac{1}{r}+p_1(r_{,1})^2+p_2r_{,1}r_{,2}\right]rd\theta$$

$$=\lim_{\varepsilon\to 0}\frac{1}{8\pi G(1-v)}\int_0^{\alpha}\left[p_1(3-4v)\,ln\frac{1}{\varepsilon}+p_1cos^2\theta+p_2\,cos\,\theta\,sin\,\theta\right]\varepsilon d\theta \qquad (2.40)$$

in a spherical coordinate system. Applying the mean value theorems for integrals, Equation (2.40) can be written:

$$\lim_{\varepsilon\to 0}\frac{3-4v}{8\pi G(1-v)}\int_0^{\alpha}\mathrm{p}_1\varepsilon\ln\frac{1}{\varepsilon}\mathrm{d}\theta$$

$$=\lim_{\varepsilon\to 0}\frac{3-4v}{8\pi G(1-v)}\varepsilon\ln\frac{1}{\varepsilon}\mathrm{p}_1(\mathrm{Q}'_\varepsilon)$$

$$=0 \qquad (2.41)$$

where Q'_ε is a point on the semicircle. And

$$\lim_{\varepsilon\to 0}\frac{1}{8\pi G(1-v)}\int_0^{\alpha}(p_1cos^2\theta+p_2\,cos\,\theta\,sin\,\theta)\varepsilon d\theta$$

$$=\lim_{\varepsilon\to 0}\frac{\varepsilon}{8\pi G(1-v)}\left[\mathrm{p}_1(\mathrm{Q}'_\varepsilon)\int_0^{\alpha}\cos^2\theta\mathrm{d}\theta+\mathrm{p}_2(\mathrm{Q}'_\varepsilon)\int_0^{\alpha}\cos\theta\sin\theta\,\mathrm{d}\theta\right]$$

$$=\lim_{\varepsilon\to 0}\frac{\varepsilon}{16\pi G(1-v)}\left[\mathrm{p}_1(\mathrm{Q}'_\varepsilon)\left(\alpha+\frac{1}{2}\sin 2\alpha\right)+\mathrm{p}_2(\mathrm{Q}'_\varepsilon)\sin^2\alpha\right]$$

$$=0 \qquad (2.42)$$

Thus, the value of Equation (2.40) is zero.

The same value of the limit of the above integral as $\varepsilon\to 0$ is obtained in the case when $l=2$. Hence,

$$\lim_{\varepsilon\to 0}\int_{C_\varepsilon} u_{lk}^{*}(P',Q')p_k(Q')\,dC(Q')=0 \qquad (2.43)$$

For the second part of the second integral, note that $\frac{\partial r}{\partial n}=1$ and $\frac{\partial r}{\partial x_l}n_k-\frac{\partial r}{\partial x_k}n_l=\frac{\partial r}{\partial x_l}\frac{\partial x_k}{\partial r}-\frac{\partial r}{\partial x_k}\frac{\partial x_l}{\partial r}=0$, when $\varepsilon\to 0$, there is:

$$\lim_{\varepsilon\to 0}\int_{C_\varepsilon} u_k(Q')\cdot p^*_{lk}(P',Q')dC(Q')$$

$$=\lim_{\varepsilon\to 0}\int_{C_\varepsilon} -\frac{u_k}{4\pi(1-v)r}\{\frac{\partial r}{\partial n}\left[(1-2v)\delta_{lk}+2r_{,l}r_{,k}\right]$$

$$-(1-2v)\left(r_{,l}n_k-r_{,k}n_l\right)\}dC$$

$$=\lim_{\varepsilon\to 0}\int_{C_\varepsilon} -\frac{u_k}{4\pi(1-v)r}\left\{\left[(1-2v)\delta_{lk}+2r_{,l}r_{,k}\right]\right\}dC. \tag{2.44}$$

When $l=1$,

$$\lim_{\varepsilon\to 0}\int_{C_\varepsilon} u_k(Q')\cdot p^*_{1k}(P',Q')dC(Q')$$

$$=\lim_{\varepsilon\to 0}\left\{-\frac{1}{4\pi(1-v)}\int_0^\alpha[u_1(1-2v)+2u_1cos^2\theta+2u_2\,cos\,\theta\,sin\,\theta]d\theta\right\}$$

$$=-\frac{1}{4\pi(1-v)}\left\{u_1\left[2(1-v)\alpha+\frac{1}{2}\sin 2\alpha\right]+u_2sin^2\alpha\right\}. \tag{2.45}$$

When $l=2$,

$$\lim_{\varepsilon\to 0}\int_{C_\varepsilon} u_k(Q')\cdot p^*_{2k}(P',Q')dC(Q')$$

$$=\lim_{\varepsilon\to 0}\left\{-\frac{1}{4\pi(1-v)}\int_0^\alpha[2u_1\,sin\,\theta\,cos\,\theta+u_2(1-2v)+2u_2sin^2\theta]d\theta\right\}$$

$$=-\frac{1}{4\pi(1-v)}\left\{u_1sin^2\alpha+u_2\left[2(1-v)\alpha-\frac{1}{2}\sin 2\alpha\right]\right\}. \tag{2.46}$$

Hence,

$$\lim_{\varepsilon\to 0}\int_{C_\varepsilon} u_k(Q')\cdot p^*_{lk}(P',Q')dC(Q')$$

$$=\begin{pmatrix}\lim_{\varepsilon\to 0}\int_{C_\varepsilon}u_k(Q')\cdot p^*_{1k}(P',Q')dC(Q')\\ \lim_{\varepsilon\to 0}\int_{C_\varepsilon}u_k(Q')\cdot p^*_{2k}(P',Q')dC(Q')\end{pmatrix}$$

$$=-\frac{1}{4\pi(1-v)}\begin{pmatrix}u_1\left[2(1-v)\alpha+\frac{1}{2}\sin 2\alpha\right]+u_2sin^2\alpha\\ u_1sin^2\alpha+u_2\left[2(1-v)\alpha-\frac{1}{2}\sin 2\alpha\right]\end{pmatrix}. \tag{2.47}$$

Substituting Equation (2.43) and (2.47) into Equation (2.38), it becomes the boundary integral equation when $\varepsilon \to 0$

$$C_{lk}(P')u_k(P') = \int_C [u^*_{lk}(P', Q')p_k(Q') - u_k(Q') \cdot p^*_{lk}(P', Q')]dC(Q'), \tag{2.48}$$

where $C_{lk}(P')$ is a constant which depends on the geometry of the surface at the point *P'*:

$$C_{lk}(P') = \frac{1}{4\pi(1-v)} \times$$

$$\begin{pmatrix} 4\pi(1-v) - \left\{2(1-v)\alpha + \frac{1}{2}\sin 2\alpha\right\} & -sin^2\alpha \\ -sin^2\alpha & 4\pi(1-v) - \left\{2(1-v)\alpha - \frac{1}{2}\sin 2\alpha\right\} \end{pmatrix} \tag{2.49}$$

For a smooth boundary surface where $\alpha = \pi$,

$$C_{lk}(P') = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}. \tag{2.50}$$

2.1.4.4 Discretization and Solutions

The goal of discretization is to divide the smooth boundary C into n segments $C_j, j = 1,2,\dots,n$. The chords between two adjacent partition points are the boundary elements (Kythe, 1995). By considering of the accuracy of calculation, there are constant, linear, quadratic and higher order boundary elements in the BEM. Different type of boundary elements will have different nodes. A constant element, as Figure 2.6 shows, has only one node, also called mid-node, which is taken at the mid-point of each element. This mid-node will represent the boundary element in order to get the known and unknown values of *u* and *p* according to the prescribed boundary conditions. For this thesis, only the constant elements will be chosen to give a detailed explanation of how BEM applied, so that the values of *u* and *p* are assumed to be constant on each element and equal to their values at its mid-nodes.

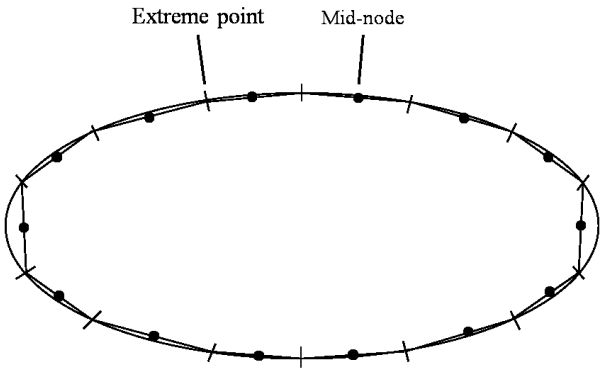


Figure 2.6: Constant elements.

For a given node P_i' on the boundary element C_i, the boundary integral equation (2.48) can be discretized to:

$$Cu_i + \sum_{j=1}^{n} (\int_{C_j} P^*(P_i', Q') d\mathrm{C}(Q')) u_j$$

$$= \sum_{j=1}^{n} (\int_{C_j} U^*(P_i', Q') d\mathrm{C}(Q'))\, p_j, \tag{2.51}$$

where

$$C = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}, \tag{2.52}$$

$$P^*(P_i', Q') = \begin{pmatrix} p_{11}^*(P_i', Q') & p_{12}^*(P_i', Q') \\ p_{21}^*(P_i', Q') & p_{22}^*(P_i', Q') \end{pmatrix},$$

$$U^*(P_i', Q') = \begin{pmatrix} u_{11}^*(P_i', Q') & u_{12}^*(P_i', Q') \\ u_{21}^*(P_i', Q') & u_{22}^*(P_i', Q') \end{pmatrix}, \tag{2.53}$$

and u_i is the nodal displacement of element C_i, u_j and p_j are the nodal displacements and tractions in the element C_j.

Let

$$\left.\begin{aligned}\widehat{H}_{ij} &= \int_{C_j} P^*(P_i', Q')dC(Q') \\ G_{ij} &= \int_{C_j} U^*(P_i', Q')dC(Q')\end{aligned}\right\}, \tag{2.54}$$

then Equation (2.51) becomes

$$Cu_i + \sum_{j=1}^{n} \widehat{H}_{ij} u_j = \sum_{j=1}^{n} G_{ij} p_j. \tag{2.55}$$

With a further step, Equation (2.51) can be written as

$$\sum_{j=1}^{n} H_{ij} u_j = \sum_{j=1}^{n} G_{ij} p_j \tag{2.56}$$

where $H_{ij} = \widehat{H}_{ij} + C\delta_{ij}$. (2.57)

Moreover, the system of n equations in Equation (2.56) can be rewritten in a matrix form as

$$HU = GP \tag{2.58}$$

Note that *G* and *H* are coefficient matrixes, *U* and *P* consist N_1 known values of *u,* N_2 known values of q as well as $2N - (N_1 + N_2)$ unknowns. Reordering all the unknowns of Equation (2.58) to the left side, the Equation (2.58) can be written as

$$AX = F \tag{2.59}$$

where *A* is the coefficient matrix, and *X* is the vector of unknowns *u* and *p*.

In order to solve for the 'unknowns' vector, the coefficients H_{ij} and G_{ij} should be calculated for the next step. It is not difficult to evaluate the integrals H_{ij} and G_{ij} for the constant element case. While for higher order elements they are much more complicated to compute so that the Gauss quadrature formulas will be used.

When $i = j$, P_i' is the mid-point of the boundary element C_i, Q' is an arbitrary point on the element C_i. Where $l_i = \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$ is the length of the element C_i.

Thus the process is as follows:

- Computation of H_{ii}

  H_{ii} is a 2×2 matrix:

$$H_{ii} = \begin{pmatrix} H_{ii}^{11} & H_{ii}^{12} \\ H_{ii}^{21} & H_{ii}^{22} \end{pmatrix}. \tag{2.60}$$

Using rigid body considerations, the values of H_{ii} are as follows:

$$H_{ii} = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}. \tag{2.61}$$

- Computation of G_{ii}

  G_{ii} is a 2×2 matrix as well:

$$G_{ii} = \begin{pmatrix} G_{ii}^{11} & G_{ii}^{12} \\ G_{ii}^{21} & G_{ii}^{22} \end{pmatrix}. \tag{2.62}$$

Using Equation (2.28) and (2.54), the value of G_{ii} can be evaluated analytically as follows:

$$\begin{aligned} G_{ii}^{11} &= \frac{l_i}{8\pi G(1-\nu)}\left[(3-4\nu)\cdot\left(1-ln\frac{l_i}{2}\right)+(r_{,1})^2\right], \\ G_{ii}^{12} &= G_{ii}^{21} = \frac{l_i}{8\pi G(1-\nu)} r_{,1} r_{,2}, \\ G_{ii}^{22} &= \frac{l_i}{8\pi G(1-\nu)}\left[(3-4\nu)\cdot\left(1-ln\frac{l_i}{2}\right)+(r_{,2})^2\right]. \end{aligned} \tag{2.63}$$

When $i \neq j$, P_i' is the mid-point of the boundary element C_i, Q' is an arbitrary point on the element C_j. H_{ij} and G_{ij} are 2×2 matrices, as before:

$$H_{ij} = \begin{pmatrix} H_{ij}^{11} & H_{ij}^{12} \\ H_{ij}^{21} & H_{ij}^{22} \end{pmatrix}, \; G_{ij} = \begin{pmatrix} G_{ij}^{11} & G_{ij}^{12} \\ G_{ij}^{21} & G_{ij}^{22} \end{pmatrix}.$$

The value of H_{ij} and G_{ij} can be evaluated from Equation (2.28), (2.54) and (2.57):

$$\begin{aligned} H_{ij}^{11} &= -\frac{1}{4\pi(1-\nu)}\int_{C_i}\frac{\partial r}{\partial n}\left[1-2\nu+2\left(r_{,1}\right)^2\right]\frac{dC}{r}, \\ H_{ij}^{12} &= -\frac{1}{4\pi(1-\nu)}\int_{C_i}\left[2\frac{\partial r}{\partial n}r_{,1}r_{,2}-(1-2\nu)\left(r_{,1}n_2-r_{,2}n_1\right)\right]\frac{dC}{r}, \\ H_{ij}^{21} &= -\frac{1}{4\pi(1-\nu)}\int_{C_i}\left[2\frac{\partial r}{\partial n}r_{,2}r_{,1}-(1-2\nu)\left(r_{,2}n_1-r_{,1}n_2\right)\right]\frac{dC}{r}, \end{aligned}$$

$$H_{ij}^{22} = -\frac{1}{4\pi(1-v)}\int_{C_i}\frac{\partial r}{\partial n}\left[1-2v+2\left(r_{,2}\right)^2\right]\frac{dC}{r}, \tag{2.64}$$

and

$$G_{ij}^{11} = \frac{1}{8\pi G(1-v)}\int_{C_i}\left[(3-4v)\,ln\frac{1}{r}+(r_{,1})^2\right]dC,$$

$$G_{ij}^{12} = G_{ij}^{21} = \frac{1}{8\pi G(1-v)}\int_{C_i} r_{,1}r_{,2}dC,$$

$$G_{ij}^{11} = \frac{1}{8\pi G(1-v)}\int_{C_i}\left[(3-4v)\,ln\frac{1}{r}+(r_{,2})^2\right]dC. \tag{2.65}$$

The unknown displacements and tractions of the boundary points can be evaluated by solving Equation (2.59). After the unknown values on the boundary were computed, the displacements at an interior/exterior point can be easily evaluated by using Equation (2.37), where (2.37) can be discretized with the following equation:

$$u_i = \sum_{j=1}^{n}(G_{ij}p_j - H_{ij}u_j). \tag{2.66}$$

The stress components at an interior/exterior point can be computed from:

$$\sigma_{ij} = \int_C D_{kij}p_k dC - \int_C S_{kij}u_k dC$$

$$= \sum_{j=1}^{n} D_{kij}p_k - \sum_{j=1}^{n} S_{kij}u_k\,,\ \ i,j,k = 1,2 \tag{2.67}$$

where

$$D_{kij} = \frac{1}{4\pi(1-v)r}\left[(1-2v)\left(\delta_{ik}r_{,j}+\delta_{jk}r_{,i}-\delta_{ij}r_{,k}\right)+2r_{,i}r_{,j}r_{,k}\right], \tag{2.68}$$

$$S_{kij} = \frac{G}{2\pi(1-v)r^2}\{2\frac{\partial r}{\partial n}\left[(1-2v)\delta_{ij}r_{,k}+v\left(\delta_{ik}r_{,j}+\delta_{jk}r_{,i}\right)-4r_{,i}r_{,j}r_{,k}\right]$$

$$+2v\left(r_i r_k n_j + r_j r_k n_i\right)+(1-2v)\left(\delta_{ik}n_j+\delta_{jk}n_i+2r_{,i}r_{,j}n_k\right)$$

$$-(1-4v)\delta_{ij}n_k\}. \tag{2.69}$$

2.2 GPU Acceleration of Numerical Computing Using OpenCL

Traditionally, the application of the BEM for a certain problem is implemented by a computer program. Most often, the computer program is run on a single CPU. Depending on the quantity of the exterior points in the problem domain, the time spent for the program execution can be lengthy. For example, more exterior points to be calculated, the higher accuracy the results will be, thereby the execution will take longer. In order to obtain a much accurate result in a reasonably short time, there are currently two approaches to accelerate the BEM program: to use a powerful CPU for the program execution, or to look for other hardware that can execute the BEM program.

2.2.1 CPU Computing

A central processing unit (CPU) is an important piece of hardware in a computer system. The main function of a CPU includes carrying out instructions of a computer program and performing data processing. Currently, most CPUs within a computer consist of an arithmetic logic unit (ALU), hardware registers and a control unit. The performance of a CPU depends on several factors, such as clock rate, instructions per clock, bandwidth etc. (Henning, 2000). Among these factors, the clock rate or execution frequency, which generally given in multiples of Hertz, leads the main role of CPU's performance. However, during the past decade, it's been proven that although increasing the frequency will obtain higher CPU's performance, it has limitations due to power and heat dissipation constraints (Gaster et al., 2012). Therefore, another solution for obtaining higher performance has been proposed: to clone a single core multiple times on the chip, which lead to the creation of multi-core CPUs.

For the traditional BEM program, the most time-consuming part in the process is the computation of the stresses and displacements for the whole interior/exterior points. Mapping the BEM program to a CPU, the busiest component is the ALU. While for most CPUs even a multi-core CPU in a computer, the increased amount of ALU still cannot afford the high need of arithmetic calculations used by the BEM program.

Besides multi-core CPUs within a computer, other powerful machines such as supercomputers or large-scale computer clusters can also be used to run BEM programs, and

definitely the results will be obtained accurately and in shorter time. However, these kinds of powerful machines are hard to reach for a practical civil engineer. Moreover, the cost of using supercomputers of computer clusters will be considerable for only performing a geomechanical evaluation.

Incidentally, with the rapid development of the computational power of the modern GPU, acceleration of the program execution can be achieved through GPU computing. Based on the cost and the speed for a large-scale data-computing program, a modern GPU has great superiority over a same price CPU for the program acceleration. Thus, GPU computing can be considered a better choice for the BEM program acceleration.

2.2.2 GPU Computing

A graphics processing unit (GPU), also known as visual processing unit (VPU), is a specialized electronic circuit equipped on the video card or motherboard or certain CPUs that is used to generate 2D and 3D graphics, images and videos in its most basic form (Nickolls and Kirk, 2009). Compared to CPUs, the history of GPUs is much shorter. The term of GPU was first proposed by NVIDIA as they introduced the GeForce 256 in 1999, which was the first GPU in the world (The world's first GPU, 1999). The early GPU was a single-chip processor, which was built around the graphics pipeline, specialized in three-dimensional (3-D) applications to create lighting effects, smoke effects and transforms objects, but little else (Blythe, 2008). Over the past few years, the GPU has evolved from a fixed-function special-purpose processor into an advanced programmable processor with both application programming interface (APIs) and hardware, increasingly focusing on the programmable aspects of the GPU (Owens et al., 2008). The rapid enhancement in programmability and capability of GPU has drawn a great deal of public attention, which on the other hand, promotes the application of general-purpose computing on the GPU.

Why people want to use a GPU for general computation? There are several reasons. First: performance. The GPU is designed for manipulating and processing graphics, therefore it has few characteristics such as large computational requirements, substantial parallelism and priority on throughput over latency. With these prominent characteristics, modern GPUs perform floating-point calculations much faster than today’s CPUs. For example, NVIDIA revealed a performance

comparison between different NVIDIA GPUs and Intel CPUs (Cuda, 2015), as Figure 2.7 and 2.8 show. Figure 2.7 displays the trend of floating-point operations performance for representative NVIDIA GPUs and Intel CPUs from 2002 to 2013; and Figure 2.8 displays the trend in representative NVIDIA GPUs and Intel CPUs' increasing memory bandwidth between 2003 and 2013. From Figure 2.7, it is clear that one of the fastest CPUs in a PC (Intel Ivy Bridge) can theoretically issue around 300 billion floating-point operations per second (300 gigaflops) in double precision calculations. However, NVIDIA's Tesla K40 GPU can perform around 1450 gigaflops in double precision calculations. Moreover, the peak memory bandwidth in Figure 2.8 has increased to around 290GB/sec for NVIDIA Tesla K40 GPU, while the Intel Ivy Bridge has only reached to (CPU) 60 GB/sec. In conclusion, the modern GPU has greatly outpaced its CPU counterpart on the calculation performance.

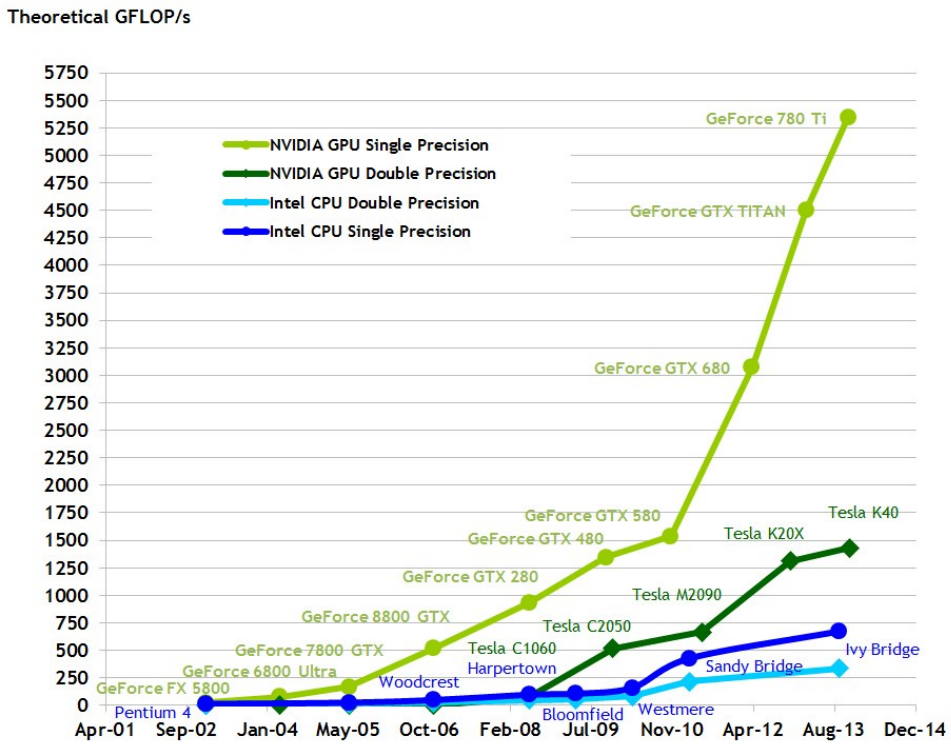


Figure 2.7: Floating-point operations per second for the CPU and GPU. (Cuda, 2015)

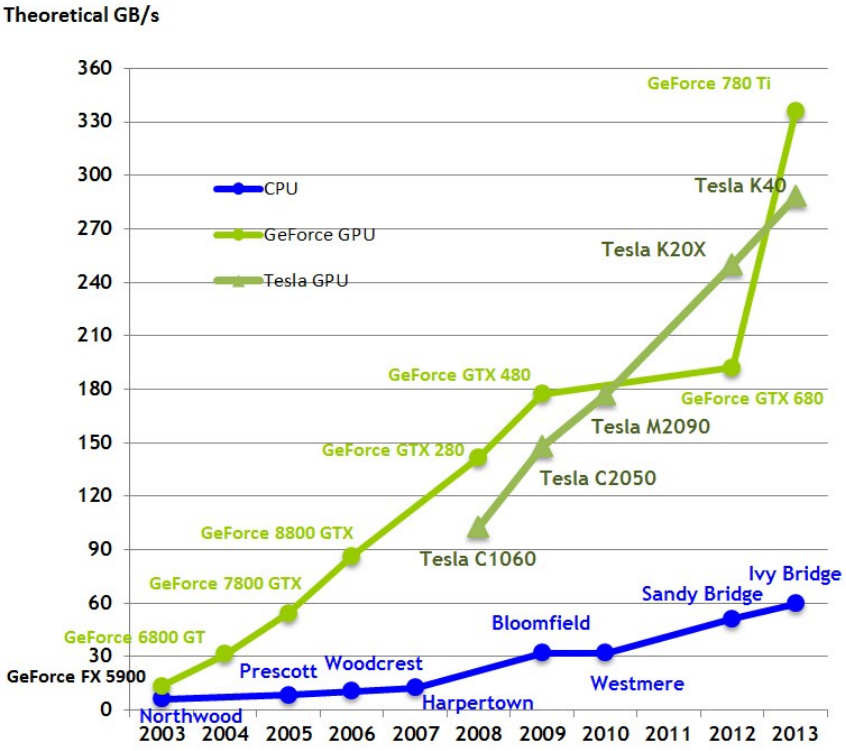


Figure 2.8: Memory bandwidth for the CPU and GPU. (Cuda, 2015)

The second reason is load balancing. A CPU is designed to carry out instructions such as performing arithmetic, logical, control and input/output operations. If the CPU is overburdened, thus limits the application performance, it's applicable to offload computational process to the GPU, which on the other hand accelerates the speed overall.

Finally: development potential. Parallelism leads the future trend of computing. In recent years, a large number of CPU applications have been ported to GPUs based on GPUs' excellent general-purpose computing performance. Looking toward the future, GPUs are on a much faster performance growth curve than CPUs. From Figure 2.7 and 2.8, a group of facts proves that: the Intel CPUs' performance over the ten years (2003 to 2013) leading up to June 2013 increased from 4 gigaflops to 300 gigaflops, or about 30 gigaflops per year. NVIDIA GPUs, however, over the same period ending with the release of the GeForce 780 Ti, increased from 8 gigaflops to 5400 gigaflops, a rate of 540 gigaflops per year. In other words, GPUs have passed CPUs in performance and will continue to outpace CPUs in the future.

How do general-purpose computing implement on the GPU? The scenario of mapping general-purpose computation onto the GPU by means of graphics hardware is intrinsically the same way as any standard graphics application. A graphics pipeline is a computational structure that GPUs take use of to process graphics. The pipeline is made up of several stages, and all geometric primitives pass through every stage. The input of the pipeline is a list of geometry, expressed as vertices in object coordinates, while the output is an image in a frame buffer. Figure 2.9 displays a typical graphics pipeline. From figure 2.9, it's clear that the vertex processor generates triangles, the rasterizer generates pixels displayed on the monitor, and fragment processor generates color for each fragment. The graphics pipeline contains four main stages: the geometry stage, the rasterization stage, the fragment stage and the composition stage (Owens et al., 2007). In the composition stage, fragments are assembled into an image of pixels.

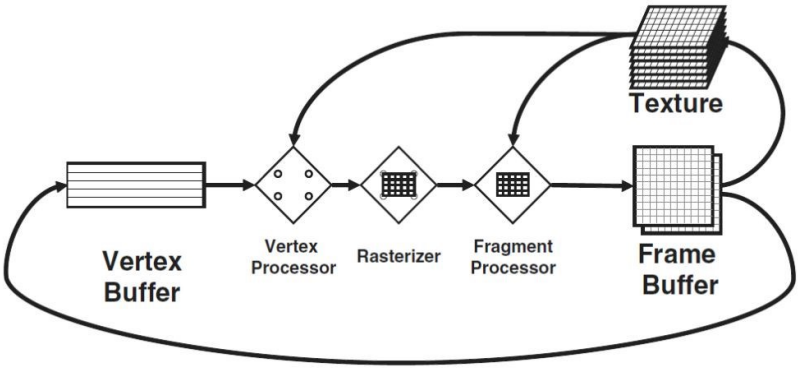


Figure 2.9: The graphic pipeline on modern GPUs. (Owens et al., 2007)

In the traditional graphics pipeline, each stage conducts fixed-function operations but does little help for general-purpose computation. Later, with the effort of graphics vendors, the fixed-function pipeline has been advanced into a more flexible programmable pipeline. The main change on this programmable pipeline is that fixed-function programs of the vertex stage and the fragment stage are replaced with user-defined programs.

To conduct general-purpose computation on GPU, a graphics API and a stream-programming model are needed. The stream-programming model will structure data into streams and express computation as arithmetic kernels that operate on streams. Using the graphics pipeline architecture, the GPGPU programming model can be constructed as follows (Owens et al., 2007):

1) Extract one or few parallel part(s) from the general-purpose program; this part is the kernel, which will execute on the GPU.
2) Specify the range of computation / the size of the output stream to invoke a kernel.
3) Use rasterizer to generate a fragment for every pixel.
4) Execute the kernel fragment program on each generated fragment.
5) The output of the kernel fragment program is a value (or vector of values) per fragment.

2.2.3 The History of GPU Computing

Before the first GPU was announced in 1999, there were only graphics accelerators equipped for PCs and computer workstations. The function of a graphics accelerator is partly separated from a CPU and that it simply accelerates graphics and it's not programmable. Represented by Geometry Engine (GE), it would perform faster rendering operations, but they are the same ones as before. As graphics accelerators were replaced by GPUs, the old concept of graphics acceleration advanced to graphics processing (Fernando et al., 2004). Starting from 1999, the first GPU GeForce 256 supported specialized graphics operations such as transform and lighting (T&L), triangle setup and clipping, of which are originally supported by CPUs. Though the rendering engines of GeForce 256 were capable of processing a minimum of 10 million polygons per second (The world's first GPU, 1999), the programmability of most GPUs at that time was still limited to graphics-specific functions. Right after the introduction of GeForce 8 series, the new generic stream-processing unit GPUs became a more generalized computing device. Based on parallel computing of GPU, two programming platforms were announced around 2006: CUDA from NVIDIA and CTM from ATI. With parallel computing on GPU, general-purpose computing on GPU has widely adopted into different fields such as scientific image processing, linear algebra, machine learning, oil exploration, statistics, 3D reconstruction and even stock options pricing determination. For example, (Harris et al., 2003) simulated cloud dynamics on graphics hardware. (Zhao et al., 2011) proposed an efficient quasi-cyclic low-density parity-check code (QC-LDPC Code, a linear error correcting code in information theory) decoder

simulator on GPU. (Krüger and Westermann, 2003) proposed a stream model of arithmetic operations in linear algebra that can be implemented on GPU.

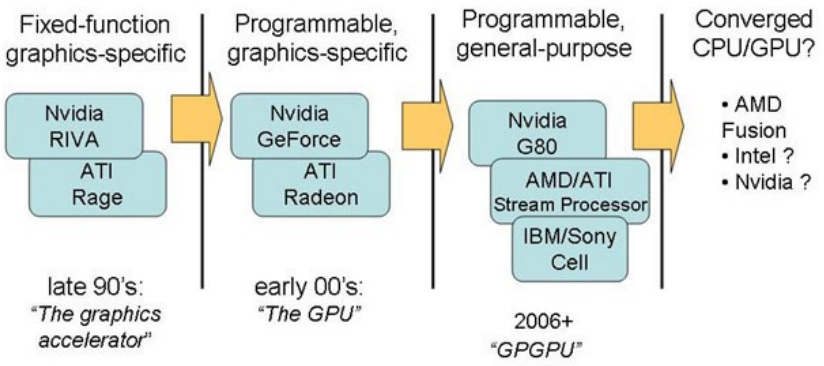


Figure 2.10: Process of GPU's development. (A (brief) history of the graphics chip: From VGA to programmable, general-purpose streaming processor, 2007)

Compared to CPU, the development of GPU is rather young, not to mention GPU computing. GPU computing was launched more than a decade ago while its application has been widely used because of its stream processing model and its parallelization computation. Multiple studies and research in different fields have proved that computing on GPUs leads a huge superiority than on CPUs. Moreover, with the rapid development on GPU's property and hardware, the GPGPU holds a vast potential for future investment.

2.2.4 Software Environments of GPGPU

At beginning, the GPGPU was carried out using fixed-function, graphics-specific units (e.g. texture filters, blending, and stencil buffer operations) through graphics APIs. In the trend of pursuing higher-level shader programming, DirectX 9 presented a C-like interface by adopting the "high-level shading language" (HLSL). Cg from NVIDIA and the OpenGL Shading Language (GLSL) were other two shading languages, which had been used substantially (Owens et al., 2008). However, programming with these shading languages was still intractable because the mentioned three were inherently shading languages, through which the computation must be expressed in graphics terms. For common programmers, higher-level languages, which were designed straightforward for computation, need to be created.

The stream-programming model was introduced as well as parallel computing when developing GPGPU. Parallelism, which is to run two or more activities in parallel, has subsequently been proved a fruitfully method for obtaining higher performance. The stream-programming model allows the streaming processor transferring data, matching the resources and communicating in parallel. Two early lauguages that treat the GPU as a streaming processor to build program models were Brook and Sh (Buck et al., 2004). Few years later, RapidMind commercialized Sh with the goal of multiple platforms including GPUs, the STI Cell broadband Engine, and multicore CPUs (Owens et al., 2008).

In the last decade, BrookGPU is no more supported by its original developers in Stanford University. Instead, AMD generalized the use of the Brook language in streaming computations. With the SW environment to include the Close to Metal (CTM) and the Compute Abstraction Layer (CAL), AMD stream programming can be operated in their highly threaded parallel architecture. NVIDIA's GPGPU programming system CUDA provides a higher lever interface than AMD's CAL. But similar to Brook, CUDA uses a C-like syntax to write programs, which compile on the GPU. Two types of parallelism, data parallel and multithreading can be achieved through CUDA while Brook only supports one dimension of parallelism, data parallelism via streaming. IBM uses message-passing-based software to take advantage of its heterogeneous, non-coherent cell architecture. Its integrated libraries which written in VHDL with C or C++ provides a prioritized environment for the FPGA system (Bacon et al., 2012).

Though each of these software environments largely enhanced the performance of programming on GPUs regarding to some specific areas, none of them made the goal of performing general-purpose computations for different hardware architectures. However, this problem can be evenly solved through the Open Computing Language (OpenCL).

2.2.5 OpenCL for GPGPU

The Open Computing Language (OpenCL) was defined and managed by the nonprofit technology consortium Khronos Group (Munshi, 2009). Actually, the OpenCL is an open standard for general purpose parallel programming across multiple processors such as CPUs, GPUs and so on. In other words, OpenCL is a heterogeneous programming framework, which consists of an applications programming interface (API) and a C-like programming language.

The language and its development environment import many conceptual frameworks from very successful, hardware specific environments such as CUDA, CAL, CTM. Combining their various characters, a hardware-independent software development environment was able to be built. The language supports different levels of parallelism and maps to multiple device systems effectively, for example, the homogeneous or heterogeneous, single- or multiple-device systems including CPUs, GPUs, FPGA and potentially other future devices (Gaster et al., 2012).

The OpenCL standard supports both data-parallel and task-parallel programming models. Using the core language and correctly following the specification, any program designed for one vendor can execute on another's hardware (Gaster et al., 2012). The host language and the device language make up the core language. The OpenCL specification is defined in four parts, or models, which are Platform model, Execution model, Memory model and Programming model (Munshi, 2009). The platform model defines relationship between the host and device. The execution model defines how the OpenCL environment is configured on the host and how kernels are executed on the device. The memory model defines the memory hierarchy for the data within the kernel. The programming model defines how the concurrency model is mapped to physical hardware (Gaster et al., 2012).

There are other existing frameworks that are used in the research of fascinating parallel computing abilities of GPUs, such like NVIDIA's CUDA and Direct Compute from Microsoft. However, CUDA is only available for NVIDIA's GPUs, so that the acceleration method of BEM cannot be applied for ATI GPUs or other powerful devices. On the other hand, though Direct Compute support graphics cards from multiple vendors, it is only specific to Microsoft Windows, therefore it is not portable between host Operating Systems. Compared to these frameworks, OpenCL provides portability across various GPU devices, OS software, and multi-core processors (Du et al., 2012). OpenCL's cross-platform, industry-wide support makes it an excellent programming model for developers to learn and use. That's the reason that OpenCL is used in this thesis, to adapt the parallel computing acceleration method of BEM to multiple GPU devices, and different operating systems.

3 Methodology -- BEM Program Implementation on CPU and the Development of the GPU Acceleration Algorithm

The practical implementation of the BEM solution running on both a serial processor (CPU) and a parallel processor (GPU) will be developed in this chapter. First, the serial implementation will be discussed, in reference to the previous chapter. The salient equations will be repeated and the corresponding code segments will be shown. Subsequently, the parallel implementation of the BEM algorithm will be presented, with references to the serial one, and a detailed explanation will be given to the usage of OpenCL. Concepts such as the data-parallel model, arithmetic optimization, and CPU-GPU transformation will be treated as well. After that, the parallel implementation, which runs on a GPU, will be described in detail.

For the computer implementation of the serial algorithm, a program *Serial.cpp* was created to solve two-dimensional linear elastic boundary value problems. This program (see Appendix 1 for the source code) is the serial program, developed using traditional principles of programming, which executes on a CPU. In order to accelerate this program, another program, *GPU.cpp* (see Appendix 1 for the source code), was developed, which executes on a GPU. This program explores the parallel algorithm by using the OpenCL framework. Both of these programs use a common input file to compute the solution values of the displacement components and stresses components at exterior points. Similarly, both programs write out an output file, where the solution values will be recorded.

3.1 Implementation of the BEM Algorithm on a CPU

The flow of the serial program can be explained using the following flow chart, as shown in Figure 3.1:

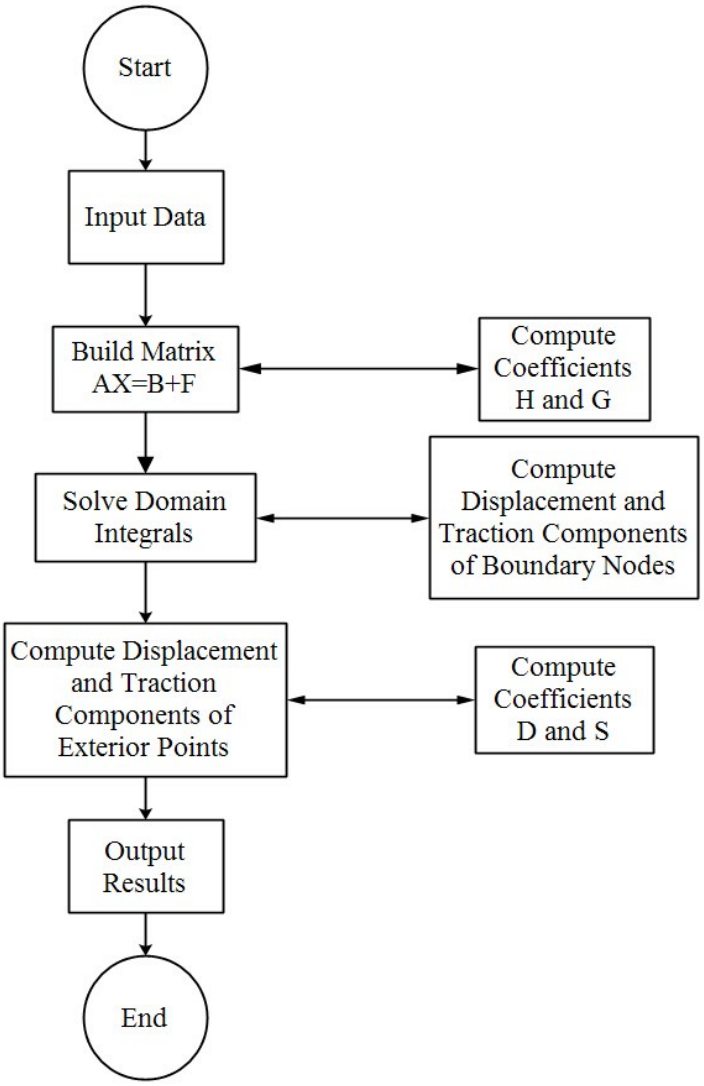


Figure 3.1: Procedure of the BEM algorithm implementation on CPU.

The serial program was implemented in the style of Kythe (1995). It contains a main function and several subroutine functions. The main function starts the program and calls the following subroutine functions: `Sys11, Inter11, and Solve.` `Sys11` is the third step in the process as the flow chart (Figure 3.1) shows. It builds the matrix AX=B+F and it calls the subroutine functions `Quad11` and `Diag11` to compute the coefficient H and G. `Solve` is the fourth step in the process. It solves the matrix equations AX=B+F thus the unknown values of displacements and tractions of the boundary points are evaluated. `Inter11` is the fifth step in the process, which computes two displacement components and three traction components of each

exterior point by using the data of the boundary points and by calling the subroutine functions `Quad11` and `Stress` to calculate the coefficients H, G and D, S.

Table 3.1: The definition of variables in the BEM program, after (Kythe, 1995).

Variable	Interpretation
N	Number of boundary elements (same as the number of nodes in this constant element case)
L	Number of exterior points where results are to be computed
M	Number of different surfaces 1 through 5
Last	Number of the last node on each different surface.
mu	Shear modulus μ
nu	Poisson's ratio ν
X, Y	Coordinates of extreme points of the elements
Xm,Ym	Coordinates of the mid-nodes
G	Matrix defined in HU = B+GP. After boundary conditions are applied, the matrix A of AX = B+F is stored in this location.
H	Matrix defined in HU = B+GP
Code	Code=0 if displacements are prescribed, Code=1 if tractions are prescribed
Bc	Prescribed boundary conditions
F	Vector defined in AX = B+F, After solution, the values of the unknowns are located here
Xi,Yi	Coordinates of the interior points
Dim	Maximum dimension of the system of HU = B+GP
Perp	Perpendicular distance from the point (x_p,y_p) to the element j
Xg,Yg	(x,y)-coordinates of Gauss points ζ_i , i=1,2,3,4
HL	Half-length of the element C_i(=Li/2)
nx,ny	n_x, n_y (components of the unit normal vector **n**)
rx,ry,rn	$r_{,x}$, $r_{,y}$, $r_{,n}$ (note that $r_{,n} = r_{,x}n_x + r_{,y}n_y$)

The subroutine `Sys11` can be summarized by the following algorithm:

1. Compute the coordinates (Xm,Ym), m=1, ... , N, which are the mid-nodes of the extreme nodes (xi,yi):

$$x_{n+1} = x_1, y_{n+1} = y_1;$$

$$X_m = \frac{x_i + x_{i+1}}{2}, Y_m = \frac{x_i + x_{i+1}}{2} \text{ for i=1 to N.}$$

The code is as follows:

```
X[N+1]=X[1];
Y[N+1]=Y[1];
for (i=1;i<=N;i++) {
        Xm[i]=(X[i]+X[i+1])*0.5;
        Ym[i]=(Y[i]+Y[i+1])*0.5;
}
```

2. Compute the matrices H and G:

for i = j, call Quad11;

i ≠ j, call Diag11.

The code is as follows:

```
if (i-j) {
        Quad11(Xm[i],Ym[i],X[j],Y[j],X[kk],Y[kk],&H[2*i-1][2*j-1],
                        &H[2*i-1][2*j],&H[2*i][2*j-1],&H[2*i][2*j],
                         &G[2*i-1][2*j-1],&G[2*i-1][2*j],&G[2*i][2*j]);
                G[2*i][2*j-1]=G[2*i-1][2*j];
        }
else {
        Diag11(X[j],Y[j],X[kk],Y[kk],&G[2*i-1][2*j-1],
     &G[2*i-1][2*j],&G[2*i][2*j]);
        H[(2*i-1)][(2*j-1)]=0.5;
        H[(2*i)][(2*j)]=0.5;
        H[(2*i-1)][(2*j)]=0.0;
```

```
            H[(2*i)][(2*j-1)]=0.0;
            G[(2*i)][(2*j-1)]=G[(2*i-1)][(2*j)];
    }
```

3. Build the matrix AX = B + F :

for *j=1* to *NN*

if *Code(j) = 1*, then for *i = 1* to *NN*

$$temp = G_{ij};\ G_{ij} = -H_{ij};\ H_{ij} = -temp = -G_{ij}$$

if *Code(j) = 0*, then for *i = 1* to *NN*

$$G_{ij} = \mu,$$

where *NN=2N*; *temp* is a temporary memory location; and if *Code=1*, the displacements were prescribed; if *Code=0*, the tractions were prescribed in the input file. As a result, the matrix A is in location G; F is not yet evaluated, but all known terms are in the location of H and U. The location H contains both known G and H, and the location U contains both known displacement values (U) and traction values (P).

The code is as follows:

```
for (j=1;j<=NN;j++) {
        if (Code[j]>0) {
                for (i=1;i<=NN;i++)   {
                        temp=G[i][j];
                        G[i][j]=-H[i][j];
                        H[i][j]=-temp;
                }
        }
        else {
                for (i=1;i<=NN;i++) {
                        G[i][j]*=mu;
                }
        }
```

4. Finally,

$$F_i = 0.0 \qquad \text{for } i = 1 \text{ to } NN$$

$$F_i = F_i + H_{ij} \cdot Bc_j \qquad \text{for } j = 1 \text{ to } NN$$

The right side vector F now stores all known values in *F*.

The code is as follows:

```
for(i=1;i<=NN;i++) {
        F[i]=0.0;
        for (j=1;j<=NN;j++)   {
                F[i]+=H[i][j]*Bc[j];
        }
```

The subroutine `Quad11` computes the off-diagonal elements of H and G by using the four-point Gauss quadrature formula.

In order to compute H_{ij} and G_{ij} ($i \neq j$), two different nodes i and j are taken into consideration. (x_p,y_p) are the coordinates of the node i; ζ_k , k=1,2,3,4 are the Gauss points marked on the element with mid-node j;ζ=1 andζ=-1 are the extreme points with coordinates (x_j, y_j) and (x_{j+1}, y_{j+1}) respectively of the node j; m denotes the slope of the boundary element with node j; and `Ra` is the distance from node i to a Gauss pointζ_k , k=1,2,3,4. The coordinates of the Gauss points are denoted by (`Xg`, `Yg`). Denote `Ax`= (x_{j+1}-x_j)/2, `Ay`= (y_{j+1}-y_j)/2, `Bx`= (x_{j+1}+x_j)/2, and `By`= (y_{j+1}+y_j)/2, then m= (y_{j+1}-y_j)/ (x_{j+1}-x_j) =`Ay/Ax`=`slope` of the element $\tilde{C}_j$.

Or in a code form:

```
Ax=(X2-X1)*0.5f;
Bx=(X2+X1)*0.5f;
Ay=(Y2-Y1)*0.5f;
By=(Y2+Y1)*0.5f;
nx=(Y2-Y1)/(2.0f*sqrtf(Ax*Ax+Ay*Ay));
```

```
ny=(X1-X2)/(2.0f*sqrtf(Ax*Ax+Ay*Ay));
slope=Ay/Ax;
```

The equation of the element with node *j* is:

$$m(x_j - x) - (y_j - y) = 0, \tag{3.1}$$

where the distance

$$\mathrm{Ra} = \sqrt{(\mathrm{x_p} - \mathrm{Xg})^2 + (\mathrm{y_p} - \mathrm{Yg})^2}, \tag{3.2}$$

and the half-length of the element $\tilde{\mathrm{C}}_\mathrm{i}$:

$$\frac{\mathrm{L_i}}{2} = \sqrt{\mathrm{Ax}^2 + \mathrm{Ay}^2} = \mathrm{HL}. \tag{3.3}$$

Denote `Denom` $= 4\pi(1-v)$, a perpendicular distance from node *i* to the element *j* is given as

`perp` $= \frac{|\mathrm{slope}\cdot x_p - y_p + (y_1 - \mathrm{slope}\cdot x_1)|}{\sqrt{\mathrm{slope}^2+1^2}}$ if `slope` exist;

or `perp`$= |x_p - x_1|$ if `slope` does not exist.

Also, the directional derivative of $\ln\left(\frac{1}{r}\right) = -\frac{1}{2}\ln(x^2+y^2)$ in the direction of $\hat{\mathrm{n}}$ is given by

$$\mathrm{D}_{\hat{\mathrm{n}}}\ln\left(\frac{1}{\mathrm{r}}\right) = \nabla \mathrm{f}\cdot\hat{\mathrm{n}} = -\frac{y}{x^2+y^2} = -\frac{r_{,x}n_x + r_{,y}n_y}{r^2} = -\frac{\mathrm{rx}\cdot\mathrm{nx}+\mathrm{ry}\cdot\mathrm{ny}}{(\mathrm{Ra})^2}, \tag{3.4}$$

where $\mathrm{rx} = (\mathrm{Xg} - \mathrm{x_p})/\mathrm{Ra} = \cos\alpha$, $\mathrm{ry} = (\mathrm{Yg} - \mathrm{x_p})/\mathrm{Ra} = \sin\alpha$.

Then, by the four-point Gauss quadrature, G_{ij} and H_{ij} can be computed as:

$$\mathrm{G}_{ij}^{11} = \int_{\tilde{\mathrm{C}}_\mathrm{j}} \mathrm{u}^*\,\mathrm{ds} = \frac{HL}{2\cdot\mathrm{Denom}\cdot\mu}\sum_{\mathrm{i}=1}^{4}[(3-4v)\ln\left(\frac{1}{\mathrm{Ra}}\right) + \mathrm{rx}^2]\mathrm{W_i}\,, \tag{3.5}$$

$$G_{ij}^{12} = G_{ij}^{21} = \frac{HL}{2\cdot\mathrm{Denom}\cdot\mu}\sum_{\mathrm{i}=1}^{4}(\mathrm{rx}\cdot\mathrm{ry})W_i, \tag{3.6}$$

$$\mathrm{G}_{ij}^{22} = \frac{HL}{2\cdot\mathrm{Denom}\cdot\mu}\sum_{\mathrm{i}=1}^{4}[(3-4v)\ln\left(\frac{1}{\mathrm{Ra}}\right) + \mathrm{ry}^2]\mathrm{W_i}, \tag{3.7}$$

and

$$\mathrm{H}_{ij}^{11} = \int_{\tilde{\mathrm{C}}_j} \mathrm{p}^* \, \mathrm{ds} = -\frac{\mathrm{perp}\cdot\mathrm{HL}}{2\cdot\mathrm{Denom}\cdot\mu} \sum_{\mathrm{i}=1}^{4} [1 - 2v + 2 \cdot rx^2] \frac{W_i}{(\mathrm{Ra})_{\mathrm{i}}^2}, \tag{3.8}$$

$$H_{ij}^{12} = -\frac{\mathrm{HL}}{2\cdot\mathrm{Denom}\cdot\mu} \sum_{\mathrm{i}=1}^{4} [\frac{2\cdot\mathrm{perp}}{(\mathrm{Ra})_i} (\mathrm{rx}\cdot\mathrm{ry}) - (1 - 2v)(\mathrm{rx} \cdot \mathrm{ny} - \mathrm{ry} \cdot \mathrm{nx})] \frac{W_i}{(\mathrm{Ra})_i}, \tag{3.9}$$

$$H_{ij}^{21} = -\frac{\mathrm{HL}}{2\cdot\mathrm{Denom}\cdot\mu} \sum_{\mathrm{i}=1}^{4} [\frac{2\cdot\mathrm{perp}}{(\mathrm{Ra})_i} (\mathrm{rx}\cdot\mathrm{ry}) - (1 - 2v)(\mathrm{ry} \cdot \mathrm{nx} - \mathrm{rx} \cdot \mathrm{ny})] \frac{W_i}{(\mathrm{Ra})_i}, \tag{3.10}$$

$$\mathrm{H}_{ij}^{22} = -\frac{\mathrm{perp}\cdot\mathrm{HL}}{2\cdot\mathrm{Denom}\cdot\mu} \sum_{\mathrm{i}=1}^{4} [1 - 2v + 2 \cdot \mathrm{ry}^2] \frac{W_i}{(\mathrm{Ra})_{\mathrm{i}}^2}. \tag{3.11}$$

Or in a code form:

```
for (i=1;i<=4;i++) {
    Xg[i]=Ax*Z[i]+Bx;
    Yg[i]=Ay*Z[i]+By;
    Ra=sqrtf((Xp-Xg[i])*(Xp-Xg[i])+(Yp-Yg[i])*(Yp-Yg[i]));
    rx=(Xg[i]-Xp)/Ra;
    ry=(Yg[i]-Yp)/Ra;
    (*G11)+=((3.0f-4.0f*nu)*logf(1.0f/Ra)+rx*rx)*W[i]*HL/(2.0f*Denom*mu);
    (*G12)+=rx*ry*W[i]*HL/(2.0f*Denom*mu);
    (*G22)+=((3.0f-4.0f*nu)*logf(1.0f/Ra)+ry*ry)*W[i]*HL/(2.0f*Denom*mu);
    (*H11)-=Perp*((1.0f-2.0f*nu)+2.0f*rx*rx)/(Ra*Ra*Denom)*W[i]*HL;
    (*H12)-=(Perp*2.0f*rx*ry/Ra+(1.0f-2.0f*nu)*(nx*ry-ny*rx))*W[i]*HL/(Ra*Denom);
    (*H21)-=(Perp*2.0f*rx*ry/Ra+(1.0f-2.0f*nu)*(ny*rx-nx*ry))*W[i]*HL/(Ra*Denom);
    (*H22)-=Perp*((1.0f-2.0f*nu)+2.0f*ry*ry)*W[i]*HL/(Ra*Ra*Denom);
}
```

The subroutine `Diag11` computes the diagonal elements G_{ii} of the matrix G, given by the following equations:

$$G_{ii}^{11} = \frac{L_i}{8\pi\mu(1-v)} \left[(3 - 4v)(1 - lnL_i) + \frac{r_1^2}{L_i^2} \right], \tag{3.12}$$

$$G_{ii}^{12} = \frac{L_i \cdot r_1 r_2}{8\pi\mu(1-v)} = G_{ii}^{21}, \tag{3.13}$$

$$G_{ii}^{22} = \frac{L_i}{8\pi\mu(1-v)} \left[(3 - 4v)(1 - lnL_i) + \frac{r_2^2}{L_i^2} \right], \tag{3.14}$$

where $L_i = \sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}$ = SR is the length of the element *i*.

The corresponding code is as follows:

```
Ax=(X2-X1)*0.5f;
Ay=(Y2-Y1)*0.5f;
SR=sqrtf(Ax*Ax+Ay*Ay);
Denom=4.0f*pi*mu*(1.0f-nu);
(*G11)=SR*((3.0f-4.0f*nu)*(1.0f-logf(SR))+(X2-X1)*(X2-X1)/(4*SR*SR))/Denom;
(*G22)=SR*((3.0f-4.0f*nu)*(1.0f-logf(SR))+(Y2-Y1)*(Y2-Y1)/(4*SR*SR))/Denom;
(*G12)=(X2-X1)*(Y2-Y1)/(4.0f*SR*Denom);
```

The subroutine `Solve` uses the Gaussian elimination method to solve the linear systems of equations *AX = B+F*. Where the parameter `found` is a flag that used to provide interchange of rows when a zero diagonal element is present (pivoting). The solutions of the linear equations systems are stored in the array `F[]` when they return to the main function.

The subroutine `Inter11` computes the values of displacement components (u_x, u_y) and stress components $(\sigma_x, \tau_{xy}, \sigma_y)$ at exterior points by using Equation (2.66) and (2.67), which are

$$u_i = \sum_{j=1}^{n} G_{ij} p_j - \sum_{j=0}^{n} H_{ij} u_j \tag{3.15}$$

and

$$\sigma_{ij} = \sum_{j=1}^{n} D_{kij} p_k - \sum_{j=1}^{n} S_{kij} u_k. \tag{3.16}$$

The function `Inter11` calls the subroutine function `Quad11` to calculate the coefficients H and G when computing the displacement *u*; and it calls the subroutine function `Stress` to calculate the coefficients D and S for computing the traction *p*.

The corresponding code is:

```
for (j=1;j<=N;j++) {
    kk=j+1;
    Quad11(Xi[k],Yi[k],X[j],Y[j],X[kk],Y[kk],&H11,&H12,&H21,&H22, &G11,&G12,&G22);
    displ[2*k-1]+=F[2*j-1]*G11+F[2*j]*G12-Bc[2*j-1]*H11-Bc[2*j]*H12;
    displ[2*k]+=F[2*j-1]*G12+F[2*j]*G22-Bc[2*j-1]*H21-Bc[2*j]*H22;
```

```
        Stress(Xi[k],Yi[k],X[j],Y[j],X[kk],Y[kk],&dx11,&dy11,&dx12,&dy12,&dx22,&dy22,&s
        x11,&sy11, &sx12,&sy12,&sx22,&sy22);
        stress[3*k-2]+=F[2*j-1]*dx11+F[2*j]*dy11-Bc[2*j-1]*sx11-Bc[2*j]*sy11;
        stress[3*k-1]+=F[2*j-1]*dx12+F[2*j]*dy12-Bc[2*j-1]*sx12-Bc[2*j]*sy12;
        stress[3*k]+=F[2*j-1]*dx22+F[2*j]*dy22-Bc[2*j-1]*sx22-Bc[2*j]*sy22;
    }
```

The subroutine `Stress` computes the coefficients D and S that are required for calculating stress components of exterior points. Using four-point Gauss quadrature formula, coefficients D and S can be computed by

$$D_{x11} = \frac{l}{8\pi(1-v)}\sum_{i=1}^{4}[\mathrm{rx}(1-2v+2\mathrm{rx}^2)]\frac{W_i}{r_i}, \tag{3.17}$$

$$D_{y11} = \frac{l}{8\pi(1-v)}\sum_{i=1}^{4}[\mathrm{ry}(2v-1+2\mathrm{rx}^2)]\frac{W_i}{r_i}, \tag{3.18}$$

$$D_{x12} = \frac{l}{8\pi(1-v)}\sum_{i=1}^{4}[\mathrm{ry}(1-2v+2\mathrm{rx}^2)]\frac{W_i}{r_i}, \tag{3.19}$$

$$D_{y12} = \frac{l}{8\pi(1-v)}\sum_{i=1}^{4}[\mathrm{rx}(1-2v+2\mathrm{ry}^2)]\frac{W_i}{r_i}, \tag{3.20}$$

$$D_{x22} = \frac{l}{8\pi(1-v)}\sum_{i=1}^{4}[\mathrm{rx}(2v-1+2\mathrm{ry}^2)]\frac{W_i}{r_i}, \tag{3.21}$$

$$D_{y22} = \frac{l}{8\pi(1-v)}\sum_{i=1}^{4}[\mathrm{ry}(1-2v+2\mathrm{ry}^2)]\frac{W_i}{r_i}, \tag{3.22}$$

and

$$S_{x11} = \frac{lG}{4\pi(1-v)}\sum_{i=1}^{4}\{2\frac{h}{r}\mathrm{rx}[1-4(\mathrm{rx})^2]+4v(\mathrm{rx})^2\mathrm{nx}+2(1-2v)[1+(\mathrm{rx})^2]\mathrm{nx}-(1-4v)\mathrm{nx}\}\frac{W_i}{r_i^2}, \tag{3.23}$$

$$S_{y11} = \frac{lG}{4\pi(1-v)}\sum_{i=1}^{4}\{2\frac{h}{r}\mathrm{ry}[1-2v-4(\mathrm{rx})^2]+4v\cdot\mathrm{rx}\cdot\mathrm{ry}\cdot\mathrm{nx}+2(1-2v)\mathrm{rx}^2\cdot\mathrm{ny}-(1-4v)\mathrm{ny}\}\frac{W_i}{r_i^2}, \tag{3.24}$$

$$S_{x12} = \frac{lG}{4\pi(1-v)}\sum_{i=1}^{4}\{2\frac{h}{r}\mathrm{ry}[v-4(\mathrm{rx})^2]+2v\cdot\mathrm{rx}\cdot(\mathrm{rx}\cdot\mathrm{ny}+\mathrm{ry}\cdot\mathrm{nx})+(1-2v)(\mathrm{ny}+2\mathrm{rx}\cdot\mathrm{ry}\cdot\mathrm{nx})\}\frac{W_i}{r_i^2}, \tag{3.25}$$

$$S_{y12} = \frac{lG}{4\pi(1-v)}\sum_{i=1}^{4}\{2\frac{h}{r}\mathrm{rx}[v-4(\mathrm{ry})^2] + 2v\cdot\mathrm{ry}\cdot(\mathrm{rx}\cdot\mathrm{ny}+\mathrm{ry}\cdot\mathrm{nx}) + (1-$$
$$2v)(\mathrm{nx}+2\mathrm{rx}\cdot\mathrm{ry}\cdot\mathrm{ny})\}\frac{W_i}{r_i^2}, \quad (3.26)$$

$$S_{x22} = \frac{lG}{4\pi(1-v)}\sum_{i=1}^{4}\{2\frac{h}{r}\mathrm{rx}[1-2v-4(\mathrm{ry})^2] + 4v\cdot\mathrm{rx}\cdot\mathrm{ry}\cdot\mathrm{ny} + 2(1-2v)\mathrm{ry}^2\cdot\mathrm{nx} -$$
$$(1-4v)\mathrm{nx}\}\frac{W_i}{r_i^2}, \quad (3.27)$$

$$S_{y22} = \frac{lG}{4\pi(1-v)}\sum_{i=1}^{4}\{2\frac{h}{r}\mathrm{ry}[1-4(\mathrm{ry})^2] + 4v(\mathrm{ry})^2\mathrm{ny} + 2(1-2v)[1+(\mathrm{ry})^2]\mathrm{ny} -$$
$$(1-4v)\mathrm{ny}\}\frac{W_i}{r_i^2}. \quad (3.28)$$

The code is as follows:

```
FA=1.0f-4.0f*nu;

AL=1.0f-2.0f*nu;

Denom=4.0f*pi*(1.0f-nu);

for (i=1;i<=4;i++) {

    Xg[i]=Ax*Z[i]+Bx;

    Yg[i]=Ay*Z[i]+By;

    Ra=sqrt(SQ(Xp-Xg[i])+SQ(Yp-Yg[i]));

    rx=(Xg[i]-Xp)/Ra;

    ry=(Yg[i]-Yp)/Ra;

    (*dx11)+=(AL*rx+2*cube(rx))*W[i]*SR/(Denom*Ra);

    (*dy11)+=(2*SQ(rx)*ry-AL*ry)*W[i]*SR/(Denom*Ra);

    (*dx12)+=(AL*ry+2*(SQ(rx))*ry)/(Denom*Ra)*W[i]*SR;

    (*dy12)+=(AL*rx+2*rx*SQ(ry))/(Denom*Ra)*W[i]*SR;

    (*dx22)+=(2*rx*SQ(ry)-AL*rx)/(Denom*Ra)*W[i]*SR;

    (*dy22)+=(AL*ry+2*cube(ry))/(Denom*Ra)*W[i]*SR;

    (*sx11)+=(2*Perp/Ra*(AL*rx+nu*2*rx-4*cube(rx))+4*nu*nx*SQ(rx)+  AL*(2*nx*SQ(rx)+2*nx)-
FA*nx)*2*mu/(Denom*SQ(Ra))*W[i]*SR;

    (*sy11)+=(2*Perp/Ra*(AL*ry-4*SQ(rx)*ry)+4*nu*nx*rx*ry+AL*2*ny*SQ(rx)-
FA*ny)*2*mu/(Denom*SQ(Ra))*W[i]*SR;

    (*sx12)+=(2*Perp/Ra*(nu*ry-4*SQ(rx)*ry)+2*nu*(nx*ry*rx+ny*SQ(rx))+
AL*(2*nx*rx*ry+ny))*2*mu/(Denom*SQ(Ra))*W[i]*SR;
```

```
        (*sy12)+=(2*Perp/Ra*(nu*rx-4*rx*SQ(ry))+2*nu*(nx*SQ(ry)+ny*rx*ry)+
        AL*(2*ny*rx*ry+nx))*2*mu/(Denom*SQ(Ra))*W[i]*SR;

        (*sx22)+=(2*Perp/Ra*(AL*rx-4*rx*SQ(ry))+4*nu*ny*rx*ry+AL*2*nx*SQ(ry)-
        FA*nx)*2*mu/(Denom*SQ(Ra))*W[i]*SR;

        (*sy22)+=(2*Perp/Ra*(AL*ry+2*nu*ry-4*cube(ry))+4*nu*ny*SQ(ry)+  AL*(2*ny*SQ(ry)+2*ny)-
        FA*ny)*2*mu/(Denom*SQ(Ra))*W[i]*SR;

    }
```

This concludes the presentation of the serial algorithm's implementation.

3.2 Parallelization of the Serial Algorithm and Its Arithmetic Optimization

In order to accelerate the serial implementation, a careful analysis is needed to determine which part in the program takes relatively the most time. For the subroutine function `Sys11`, which consists of 2N loops to build the matrix AX=B+F, where 2N is the two times the number of boundary elements. As a result, a grid of 2N is to be computed. Similarly, for the subroutine function `Solve`, a number of 2N loops will be executed to solve the matrix, which is the same number as in subroutine `Sys11`. However, for the subroutine function `Inter11`, a number of L loops will be executed to compute the displacements and tractions of the exterior points, where L is the total number of the exterior points. In this case, L can be a large number, easily in excess of 1000^2, depending on how detailed and accurate the result are sought in determining the response of a rock mass far from an excavation. In other words, if it is desired to have a relatively high accuracy of the result, then a relatively large number of L should be chosen. Generally, a variable grid will be used in computing the response of a rock mass, and the grid size may be from 10^2 to 1000^2. In order to find out the relationship between the grid size and the time for the execution, a group of tests were conducted. Figure 3.2 shows the trend of processing time with the variation of exterior point size for subroutine functions `Sys11`, `Solve` and `Inter11`. From Figure 3.2, it is clear that the processing time of subroutine functions `Sys11` and `Solve` is relatively unaffected by the number of exterior points, while the processing time for subroutine function `Inter11` shows a quadratic trend with the increase of the exterior grid size. As a result, the subroutine function `Inter11` has a high possibility of taking much more time compared to other subroutine functions `Sys11` and `Solve`. Thus, the acceleration on the subroutine function `Inter11` will be the most beneficial in speeding up the BEM algorithm.

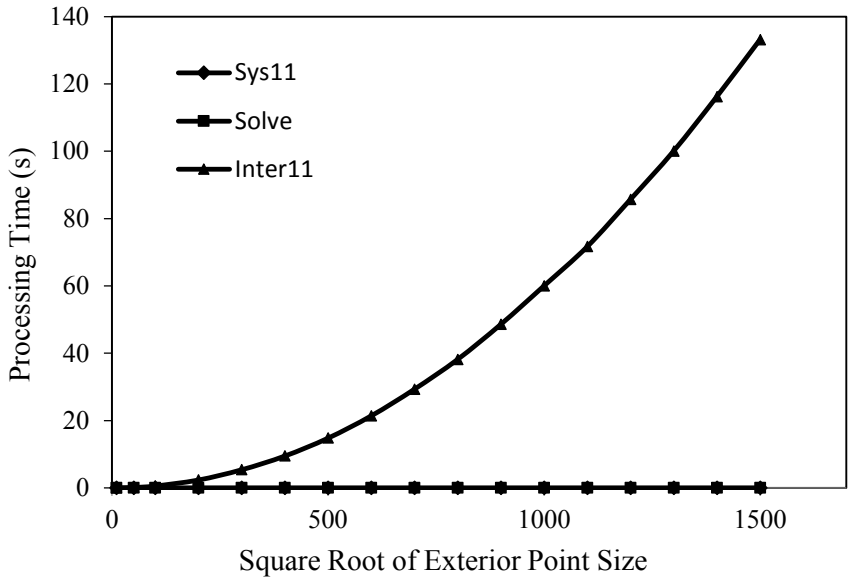


Figure 3.2: Variation of exterior point size versus processing time of subroutine functions `Sys11`, `Solve` and `Inter11`.

Though the subroutine function `Inter11` displays the data-intensive character, which fits the role of GPU Computing, the question arises if it is possible to run it efficiently in parallel on a GPU. From Equations (3.15) and (3.16), it is clear that each exterior point computed in the subroutine function `Inter11` is independent from all other points. This is the key, thus for each point on the computing grid, the process follows the same sequence: calling subroutine function `Quad11` to calculate the coefficients H and G; computing the displacements using the same equation; calling subroutine function `Stress` to calculate the coefficients D and S; and computing the tractions using the same equation. Because all the unknown values of the boundary points have already been solved in the previous subroutine function `Solve`, the coefficients H and G can be easily computed only by using the data of boundary points and the current exterior point; similarly, the coefficients D and S only depend on the data of boundary points and current exterior point. Therefore, the process of computing one exterior point is independent from computing any other exterior points. In the serial program, the exterior points can be assumed to be computed in a consecutive sequence, where the next points will not start to be computed until the previous point is finished. While in the parallel GPU program, the exterior points can be assumed to be computed in parallel, organized in large blocks. A number of blocks

can be computed at the same time. Thus, this organization of exterior point computation meets the requirements that the parallelization works well for the subroutine function `Inter11`.

In conclusion, the potentially high arithmetic intensity and high data parallelism make the GPU an attractive platform to run the subroutine function `Inter11` compared to other parts of the serial program.

3.3 Implementation of BEM Algorithm on a GPU

The GPU implementation program follows a similar procedure like the serial one, while the main difference is that the computing of the L loops in the subroutine function `Inter11` will be executed on GPU instead of CPU. The process of how the GPU implementation program executes with OpenCL can be explained using the flowchart of Figure 3.3.

In the GPU program, the main function starts the program and it calls the following subroutine functions: `Sys11`, `Solve` and `Inter11GPU`. The subroutine functions `Sys11`, `Solve` remain the same operations as in the serial program. While in the subroutine function `Inter11GPU`, the OpenCL environment is set up on the host CPU to prepare the execution of kernels on the device GPU. In the kernel there are three subroutine functions: `Helper`, `Quad11_Helper` and `Stress_Helper`. The subroutine function `Helper` is used to compute the displacements and tractions at exterior points. It calls the other subroutine functions `Quad11_Helper` and `Stress_Helper` to help compute the coefficients D and S. After that, all the results are transferred back to the host, which is the CPU, and the result will be written to the output. The following section is comprised of two parts: explanation of the OpenCL environment setup and the creation of the kernel.

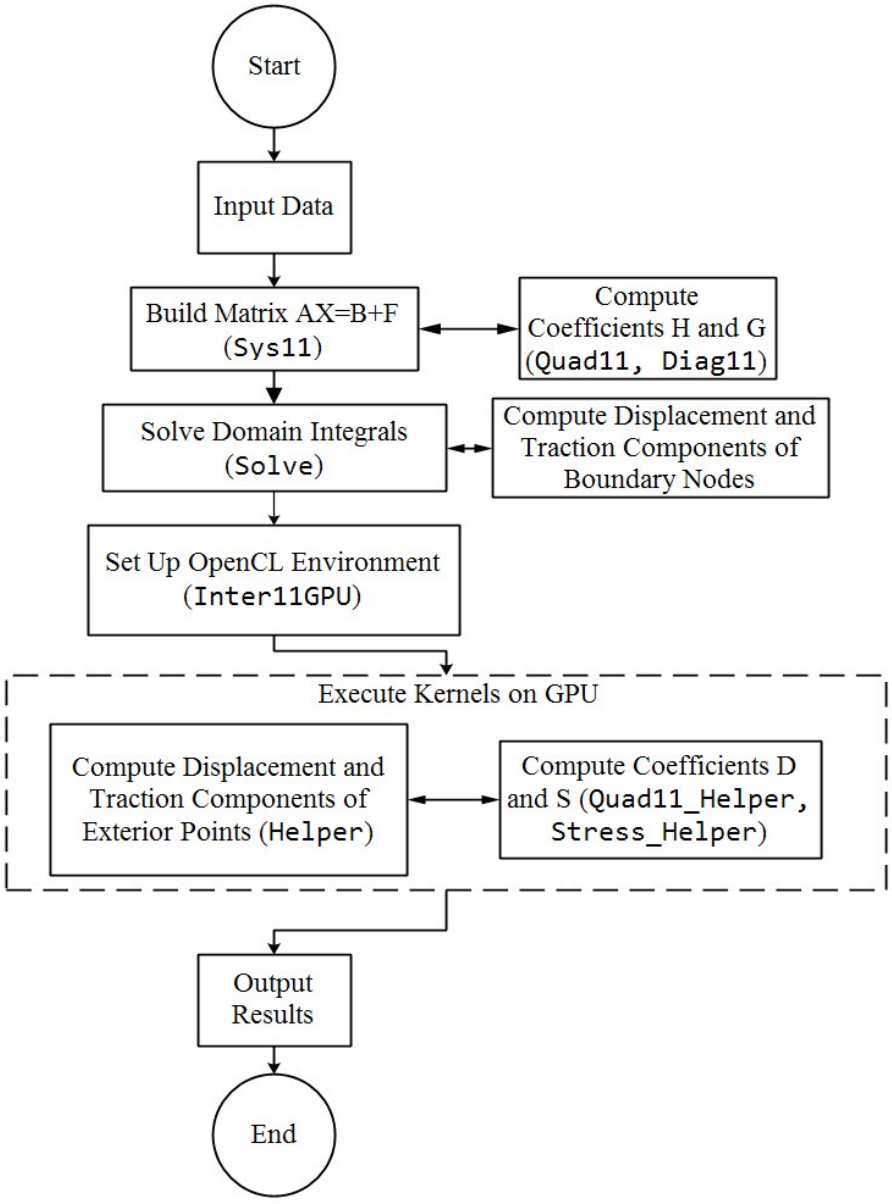


Figure 3.3: Procedure of the GPU program.

3.3.1 Explanation of the OpenCL Environment Setup

The process of setting up and executing the OpenCL environment usually follows a fixed pattern. Generally, the pattern includes 12 steps:

1) Discover and initialize the compute platforms;

2) Discover and initialize the compute devices;

3) Create an OpenCL context;

4) Create a command queue;

5) Create memory buffers;

6) Write host data to device buffers;

7) Create the program and compile it;

8) Create the kernel and set the kernel arguments;

9) Configure the work-item structure;

10) Enqueue the kernel for execution (this executes the compute kernel);

11) Read the output buffer back to the host;

12) Release OpenCL resources.

The detailed explanation of each step is as follows:

STEP 1: Discover and initialize the compute platforms

Platforms are the implementations of the OpenCL API. The first two steps can be summarized as the Platform Model. The Platform Model defines the interaction of the host and devices and provides hardware model for devices. In the Platform Model, there is a single host connected to one or more devices. In the GPU program, the CPU is the host and the GPU is the only OpenCL device. The OpenCL application submits commands from the host CPU to execute computations on the processing elements within a device GPU.

The API function `clGetPlatformIDs()` is used to discover and initialize the compute platforms. It will be called twice by the application. First, the API function `clGetPlatformIDs()` is used to obtain the list of available platforms on the system. After finding the available platforms, we can allocate space for each platform to hold the platform information. Then the function `clGetPlatformIDs()` will be called for the second time to fill in the platforms. Note that the `unsigned int` pointer `status` is a flag to check the output of each OpenCL API call. The code is as follows:

```
cl_int status;
```

```
cl_uint numPlatforms=0;
cl_platform_id* platforms=NULL;
status=clGetPlatformIDs(0,NULL,&numPlatforms);
platforms=(cl_platform_id*)malloc(numPlatforms*sizeof(cl_platform_id));
status=clGetPlatformIDs(numPlatforms,platforms,NULL);
```

STEP 2: Discover and initialize the compute devices

The API function `clGetPlatformIDs()` is used to discover the number of devices present. Similar to the API function `clGetPlatformIDs()`, the API function `clGetDeviceIDs()` will also be called twice by the application. In the first call, the number of devices present will be passed to the argument `&numDevices`. Then it will allocate enough space for each device. Then the API function `clGetDeviceIDs()` will be called for the second time to fill in the devices. Note that the `device_type` argument can be used to limit the devices to GPUs only (`CL_DEVICE_TYPE_GPU`), CPUs only (`CL_DEVICE_TYPE_CPU`), and all devices (`CL_DEVICE_TYPE_ALL`). In the GPU program, the GPU was chosen as the only OpenCL device, so the `device_type` argument was set to `CL_DEVICE_TYPE_GPU`. The corresponding code is as follows:

```
cl_uint numDevices=0;
cl_device_id* devices=NULL;
status=clGetDeviceIDs(platforms[0],CL_DEVICE_TYPE_GPU,0,NULL,&numDevices);
devices=(cl_device_id*)malloc(numDevices*sizeof(cl_device_id));
status=clGetDeviceIDs(platforms[0],CL_DEVICE_TYPE_GPU,numDevices,devices,NULL);
```

STEP 3: Create an OpenCL context

In OpenCL, a context is an abstract container that exists on the host. A context is used for coordinating the host-device interaction, for managing objects such as command-queues, memory, projects and kernel objects and for tracking the programs and kernels when executing. The API function `clCreateContext()` is used to create a context and associate it with the devices:

```
cl_context context=NULL;
context=clCreateContext(NULL,numDevices,devices,NULL,NULL,&status);
```

STEP 4: Create a command queue

A command queue is used on a host to request action by the device. Once the host decides which devices to work with and a context is created, one command queue needs to be created for each device. By submitting commands to a proper command queue, the host can communicate with an abstract device. The API function `clCreateCommandQueue()` is used to create a command queue and associate it with a device:

```
cl_command_queue queue;
queue=clCreateCommandQueue(context,devices[0],0,&status);
```

STEP 5: Create memory buffers

Memory buffer is one type of memory objects that stores data on an OpenCL device. Images are the other type of memory objects. A buffer object stores a one-dimensional collection of elements whereas an image object is used to store a two- or three-dimensional texture, frame-buffer or image. In other words, buffers are equivalent to arrays in C where data elements are stored serially in memory. Images are designed as opaque objects that improve performance like data padding and other optimizations.

Buffer objects are described by `cl_men` objects. The API function `clCreateBuffer()` is used to allocates the buffer and it returns a memory object. Note that before creating a buffer, we need to define the size of the buffer and a context in which the buffer will be allocated. Besides, the caller can supply flags that specify that the data is read-only, write-only, or read-write. In the GPU program, `d_Xi`, `d_Yi`, `d_X`, `d_Y`, `d_F` and `d_Bc` are input memory buffers because the data in these buffers are transferred to the device; `d_displ` and `d_stress` are output buffers because the data in those buffers are transferred from the device back to the host. The code is as follows:

```
// storage size for buffers
const int sizeXY=52;
const int sizeBcF=101;
size_t datasizeXY=sizeof(float)*sizeXY;
size_t datasizeBcF=sizeof(float)*sizeBcF;
size_t datastress=sizeof(float)*3*(L+1);
```

```
size_t datadispl=sizeof(float)*2*(L+1);
size_t dataXiYi=sizeof(float)*(L+1);
//Create memory buffers
cl_mem d_Xi;
cl_mem d_Yi;
cl_mem d_X;
cl_mem d_Y;
cl_mem d_F;
cl_mem d_Bc;
cl_mem d_displ;
cl_mem d_stress;
// input buffers
d_Xi=clCreateBuffer(context,CL_MEM_READ_ONLY,dataXiYi,NULL,&status);
d_Yi=clCreateBuffer(context, CL_MEM_READ_ONLY,dataXiYi,NULL,&status);
d_X=clCreateBuffer(context,CL_MEM_READ_ONLY,datasizeXY,NULL,&status);
d_Y=clCreateBuffer(context,CL_MEM_READ_ONLY,datasizeXY,NULL,&status);
d_F=clCreateBuffer(context,CL_MEM_READ_ONLY,datasizeBcF,NULL,&status);
d_Bc=clCreateBuffer(context,CL_MEM_READ_ONLY,datasizeBcF,NULL,&status);
// output buffers
d_displ=clCreateBuffer(context,CL_MEM_WRITE_ONLY,datadispl,NULL,&status);
d_stress=clCreateBuffer(context,CL_MEM_WRITE_ONLY,datastress,NULL,&status);
```

STEP 6: Write host data to device buffers

The API function `clEnqueueWriteBuffer()` is used to write host data to an OpenCL buffer. Note that the argument `blocking_write` is set to `CL_FALSE` to allow the function return before the write operation has completed. The code is as follows:

```
status=clEnqueueWriteBuffer(queue,d_Xi,CL_FALSE,0,dataXiYi,Xi,0,NULL,NULL);
status=clEnqueueWriteBuffer(queue,d_Yi,CL_FALSE,0,dataXiYi,Yi,0,NULL,NULL);
status=clEnqueueWriteBuffer(queue,d_X,CL_FALSE,0,datasizeXY,X,0,NULL,NULL);
status=clEnqueueWriteBuffer(queue,d_Y,CL_FALSE,0,datasizeXY,Y,0,NULL,NULL);
status=clEnqueueWriteBuffer(queue,d_F,CL_FALSE,0,datasizeBcF,F,0,NULL,NULL);
```

```
status=clEnqueueWriteBuffer(queue,d_Bc,CL_FALSE,0,datasizeBcF,Bc,0,NULL,NULL);
status=clEnqueueWriteBuffer(queue,d_displ,CL_FALSE,0,datadispl,displ,0,NULL,NULL);
status=clEnqueueWriteBuffer(queue,d_stress,CL_FALSE,0,datastress,stress,0,NULL,NULL);
```

STEP 7: Create and compile the program

The OpenCL C code, which is written to run on an OpenCL device, is the parallel program. An OpenCL program consists of a set of kernels that are identified as functions declared with the `__kernel` qualifier in the program source. In the GPU program, the API function `clCreateProgramWithSource()` is used to create a program object. After the program is created, it will be compiled at runtime through an API call `clBuildProgram()`. The code is as follows:

```
cl_program program;
program=clCreateProgramWithSource(context,1,&source,NULL,&status);
status=clBuildProgram(program,1,devices,NULL,NULL,NULL);
```

STEP 8: Create the kernel and set the kernel arguments

An OpenCL kernel is a function declared in a program. As described before, a kernel is identified by the `_kernel` qualifier applied to any function in a program. By extracting the kernel from the `cl_program`, a `cl_kernel` object can be used to execute kernels on a device. The API function clCreateKernel() is used to create a kernel. The name of the kernel “helper” is passed to the function, along with the program object.

```
cl_kernel kernel=NULL;
kernel=clCreateKernel(program,"helper",&status);
```

Before the kernel can be actually executed, it needs to be dispatched through an enqueue function. Thus, each kernel argument must be specified individually using a call to the API function `clSetKernelArg()`. The code is as follows:

```
status=clSetKernelArg(kernel,0,sizeof(int),&L);
status|=clSetKernelArg(kernel,1,sizeof(int),&N);
status|=clSetKernelArg(kernel,2,sizeof(float),&nu);
status|=clSetKernelArg(kernel,3,sizeof(float),&mu);
```

```
status|=clSetKernelArg(kernel,4,sizeof(cl_mem),&d_Xi);
status|=clSetKernelArg(kernel,5,sizeof(cl_mem),&d_Yi);
status|=clSetKernelArg(kernel,6,sizeof(cl_mem),&d_X);
status|=clSetKernelArg(kernel,7,sizeof(cl_mem),&d_Y);
status|=clSetKernelArg(kernel,8,sizeof(cl_mem),&d_F);
status|=clSetKernelArg(kernel,9,sizeof(cl_mem),&d_Bc);
status|=clSetKernelArg(kernel,10,sizeof(cl_mem),&d_displ);
status|=clSetKernelArg(kernel,11,sizeof(cl_mem),&d_stress);
```

STEP 9: Configure the work-item structure

The unit of concurrent execution of the kernel is a *work-item*. In other words, the kernel will be queued for execution on each work-item of the device. In the OpenCL runtime, the number of work-items is decided by the number of the input and output arrays. For the GPU program, the kernel will be executed for each one of the exterior points to compute its displacement components and stress components. Thus the number of the work-item will be set to the number of exterior points.

Since OpenCL can dispatch vast numbers of work-items and supports execution in fine-grained work-items, it is valid to have concerns about scalability. In order to achieve scalability, the work-items can be divided to smaller, equally sized workgroups. When a kernel is executed, the number of work-items is specified as an n-dimensional range, which is NDRange for short. The `global_work_size` parameter specifies the number of work-items in each dimension of the NDRange, and `local_work_size` specifies the number of work-items in each dimension of the workgroups. For example, a model with 100^2 exterior points, the code is as follows:

```
size_t globalWorkSize[1];
size_t localWorkSize[1];
globalWorkSize[0]=10000;
localWorkSize[0]=1000;
```

STEP 10: Enqueue the kernel for execution

Executing a kernel on a device requires enqueuing a command through a call to an enqueue function `clEnqueueNDRangeKernel()`. The `clEnqueueNDRangeKernel()` will return immediately after the command is enqueued in the command queue and likely before the kernel has even started execution. Thus the function `clFlush()` can be used to block execution on the host until all of the commands in a command queue have been removed from the queue. The code corresponding to this is:

```
status=clEnqueueNDRangeKernel(queue,kernel,1,NULL,globalWorkSize,localWorkSize,0,NULL,NULL
);

clFlush(queue);
```

STEP 11: Read the output buffer back to the host

Similar to the function `clEnqueueWriteBuffer()` in step 6, a call to `clEnqueueReadBuffer()` is used to transfer data back from an OpenCL buffer to the host:

```
clEnqueueReadBuffer(queue,d_displ,CL_TRUE,0,datadispl,displ,0,NULL,NULL);

clEnqueueReadBuffer(queue,d_stress,CL_TRUE,0,datastress,stress,0,NULL,NULL);
```

STEP 12: Release OpenCL resource

A series of `clRelease` functions are used to release OpecnCL resources to clean up:

```
clReleaseMemObject(d_Xi);

clReleaseMemObject(d_Yi);

clReleaseMemObject(d_X);

clReleaseMemObject(d_Y);

clReleaseMemObject(d_F);

clReleaseMemObject(d_Bc);

clReleaseMemObject(d_displ);

clReleaseMemObject(d_stress);

clReleaseKernel(kernel);

clReleaseProgram(program);

clReleaseCommandQueue(queue);

clReleaseContext(context);
```

3.3.2 Creation of the Kernel

The creation of kernel can be thought of as instance of a parallel map operation. In the GPU program, the keyword `__kernel` is added to the beginning of the functions `helper`, `Quad11_Helper` and `Stress_Helper` to declare that they are kernel functions. Similar to C functions, they have a return type of `void`. The pointers in the argument list are required to specify the address space. In the function `helper`, all input and output memory buffers are declared in global memory, which are qualified with the keyword `__global`. That is because the global memory is visible to all compute units on the device. It is required by the OpenCL API that any data transferred from the host to the device or back from the device to the host must reside in global memory. When an OpenCL device begins executing a kernel, it provides intrinsic functions to identify the work-items. The call `get_global_id(0)` is used to specify the position of each work-item. Thus, for example, the code of the function `helper` is as follows:

```
__kernel void helper(int L,
                     int N,
                     float nu,
                     float mu,
                     __global float* Xi,
                     __global float* Yi,
                     __global float* X,
                     __global float* Y,
                     __global float* F,
                     __global float* Bc,
                     __global float* displ,
                     __global float* stress)
{
    const int idx=get_global_id(0);
    int idxPlus1;
    int j,kk;
    float dx11,dy11,dx12,dy12,dx22,dy22,sx11,sy11,sx12,sy12,sx22,sy22;
    float H11,H12,H21,H22,G11,G12,G22;
    idxPlus1=idx+1;
    if (idxPlus1<L+1) {
```

```
            for (j=1;j<=N;j++) {
            kk=j+1;
            Quad11_Helper(Xi[idxPlus1],Yi[idxPlus1],X[j],Y[j],X[kk],Y[kk],nu,mu,&H11,&H12,&H21
            ,&H22,&G11,&G12,&G22);
            displ[2*idxPlus1-1]+=F[2*j-1]*G11+F[2*j]*G12-Bc[2*j-1]*H11-Bc[2*j]*H12;
            displ[2*idxPlus1]+=F[2*j-1]*G12+F[2*j]*G22-Bc[2*j-1]*H21-Bc[2*j]*H22;
            Stress_Helper(Xi[idx+1],Yi[idx+1],X[j],Y[j],X[kk],Y[kk],nu,mu,&dx11,&dy11,&dx12,&d
            y12,&dx22,&dy22,&sx11,&sy11, &sx12,&sy12,&sx22,&sy22);
            stress[3*idxPlus1-2]+=F[2*j-1]*dx11+F[2*j]*dy11-Bc[2*j-1]*sx11-Bc[2*j]*sy11;
            stress[3*idxPlus1-1]+=F[2*j-1]*dx12+F[2*j]*dy12-Bc[2*j-1]*sx12-Bc[2*j]*sy12;
            stress[3*idxPlus1]+=F[2*j-1]*dx22+F[2*j]*dy22-Bc[2*j-1]*sx22-Bc[2*j]*sy22;
            }
        }
    }
```

From the code, we can see that the function `helper` is similar to the function `Inter11` in the serial program. They both call other functions to help compute the coefficients H, G and D, S. While the difference between them is that the inner loop `for (j=1;j<=N;j++)` in the function `Inter11` has been replaced by the loop counter of work-items in the function `helper`: `idx=get_global_id(0)`. Besides, the two help functions `Quad11_Helper` and `Stress_Helper` are executed on the GPU device as well to avoid needless data transferring between the host and the device.

3.4 Conclusions

This chapter started with the introduction of the implementation of a BEM program on a CPU device of a computer. The subsequent analysis of this BEM program regarding which is the most promising part to accelerate, revealed that the execution of the subroutine function `Inter11` on the OpecnCL device GPU helps the most, since it is computation-bound. Then a detailed acceleration implementation through OpenCL functions was introduced together with setting up the OpenCL environment and the creation of the kernel.

4 GPU Acceleration Verification and Performance Analysis

This chapter will consider the results of stress analysis and investigate the performance of both the serial implementation of the BEM program (on a CPU) and the parallel implementation of the BEM program (on a GPU). As a test case, a two-dimensional linear elastic problem was used, as follows. Consider the case of a circular excavation subjected to internal pressure embedded in an infinite medium, as shown in Figure 4.1. A practical scenario that can be represented using this model would be a pressurized supply tunnel for hydro-electric power generation. The rock mass properties used were Shear Modulus of G=94500MPa, Poisson Ratio v=0.1, the internal pressure was 100kPa. The units in Figure 4.1 were in meters. The input file is given in Appendix 2.

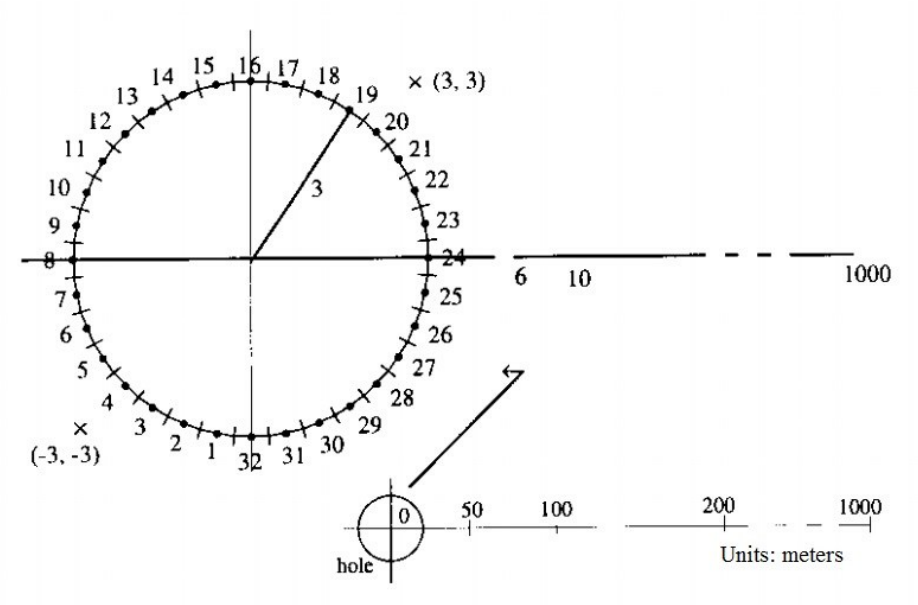


Figure 4.1: Circular excavation in an infinite medium (Kythe, 1995).

The information on the computer hardware, which implements the serial and parallel BEM code, is shown in Table 4.1. It should be noted that all sample executions of the code were done on this computer. Therefore, it is possible that the acceleration performance may be somewhat different for other CPUs and GPUs.

Table 4.1: Computing device parameters.

Device	CPU	GPU
Chipset Manufacturer:	Intel	NVIDIA
Chipset Model:	Core i3-3220	GTX650Ti
Processor Speed:	3.30GHz	941MHz
Processor Memory:	8GB	4GB

In the following sections, results will be presented for the implementation of the serial and parallel programs for this test case. After collecting the results from each program execution, comparisons will be made of the accuracy of the results between serial program and parallel program. Then, performance analysis will be conducted regarding to what degree the speedup can be made of the acceleration parallel program over the traditional serial program. Studies will also be focused on the double precision floating-point implementations of the serial program and parallel program. Comparisons of the accuracy and the performance between the single precision floating-point implementations and double precision floating-point implementations will be presented afterward. In addition, a further study regarding optimizing acceleration performance by changing the workgroup structure will be investigated.

4.1 Verification of Results

Results in the output files of serial and parallel programs were recorded in the same form, as follows:

Exterior point displacements

X_i	Y_i	Displacement $_X$	Displacement $_Y$
4.000000,	0.000000)	0.001202	0
14.333333,	0.000000)	0.000335	0
24.666666,	0.000000)	0.000195	0
4.000000,	11.666667)	0.000126	0.000369
14.333333,	11.666667)	0.000202	0.000164
24.666666,	11.666667)	0.000159	0.000075
4.000000,	23.333334)	0.000034	0.0002

(14.333333, 23.333334)	0.000092	0.00015
(24.666666, 23.333334)	0.000103	0.000097

…

Exterior point stresses

X_i Y_i	σ_X	τ_{XY}	σ_Y
(4.000000, 0.000000)	-56.834614	0.000006	56.823196
(14.333333, 0.000000)	-4.423013	0	4.423013
(24.666666, 0.000000)	-1.493453	0	1.493453
(4.000000, 11.666667)	4.717086	-3.665452	-4.717086
(14.333333, 11.666667)	-0.540047	-2.605036	0.540047
(24.666666, 11.666667)	-0.774224	-0.943421	0.774224
(4.000000, 23.333334)	1.528786	-0.540025	-1.528787
(14.333333, 23.333334)	0.547795	-1.080868	-0.547795
(24.666666, 23.333334)	-0.043754	-0.786964	0.043754

…

In order to gain valuable insight into result verification, a group of results will be collected from executions of both serial and parallel programs. By changing the grid size of exterior points, solution results will be computed for each grid size. For the circular excavation case, a set of exterior point numbers will be taken into execution. The grid size will be varied from 100^2 to 1500^2, with increments of 100^2. For each grid size, the serial and parallel program will be executed and to create an output file, which contains the solution results. Then, a comparison can be made for every exterior point between the serial program and GPU accelerated program. Taking the point (4.000000, 0.000000) in the grid of 100^2 as an example, the solution results are shown in Table 4.2:

Table 4.2: Solution results of point (4.000000, 0.000000) in the 100^2 grid.

	Displacement $_X$	Displacement $_Y$	σ_X	τ_{XY}	σ_Y
Serial	0.001202	0.000000	-56.834614	0.000006	56.823196
GPU	0.001202	0.000000	-56.834629	0.000008	56.823200
Absolute difference	0	0	0.000015	0.000002	0.000004

It can be observed from Table 4.2 that for the exterior point (4.000000, 0.000000), the absolute difference at stress σ_X is 0.000015, the absolute difference at stress τ_{XY} is 0.000002 and the absolute difference at stress σ_Y is 0.000004. Based on these comparisons, an assumption is brought out that solution values computed on CPU have little difference with the values computed on GPU. In order to prove this assumption, we need to calculate the maximum difference between the serial solution and GPU solution for all the exterior points, thus the L-infinity (L_∞) Norm measurement (Cadzow, 1973) is taken into consideration. The L_∞ Norm is defined by

$$\|x\|_\infty = max[|x(1)|, |x(2)|, \cdots, |x(n)|]. \quad (4.1)$$

In the comparison, each vector has a difference between the serial solution and the parallel solution for every single exterior point. Applying the difference into the L_∞ Norm measurement, we can calculate the maximum absolute difference for vectors Displacement $_X$, Displacement $_Y$, σ_X, τ_{XY} and σ_Y respectively for all the exterior points. Measurement is worked out through various exterior point model sizes, from 100^2 to 1500^2, and the data collected is shown in Table 4.3.

If the numbers in Table 4.3 just give an idea of the difference between the serial solutions and the parallel solutions, the bar chart in Figure 4.2 presents a more visual demonstration of the previous assumption. The bar chart in Figure 4.2 is drawn from Table 4.3. From the bar chart in Figure 4.2, we can observe that the differences in vectors Displacement $_X$ and Displacement $_Y$ are at the lowest level, which was 0.000001m, where the solution value was 0.000102m; the greatest difference belongs to the vector σ_X, which was 7.2E-05MPa, where the solution value was −53.445778MPa. Comparing the differences to the solution values, it is clear that the difference will not affect the accuracy of the solution results. To conclude, the assumption is proved that the

solution results computed from the accelerated parallel program on GPU have little difference with the traditional solution results in serial program.

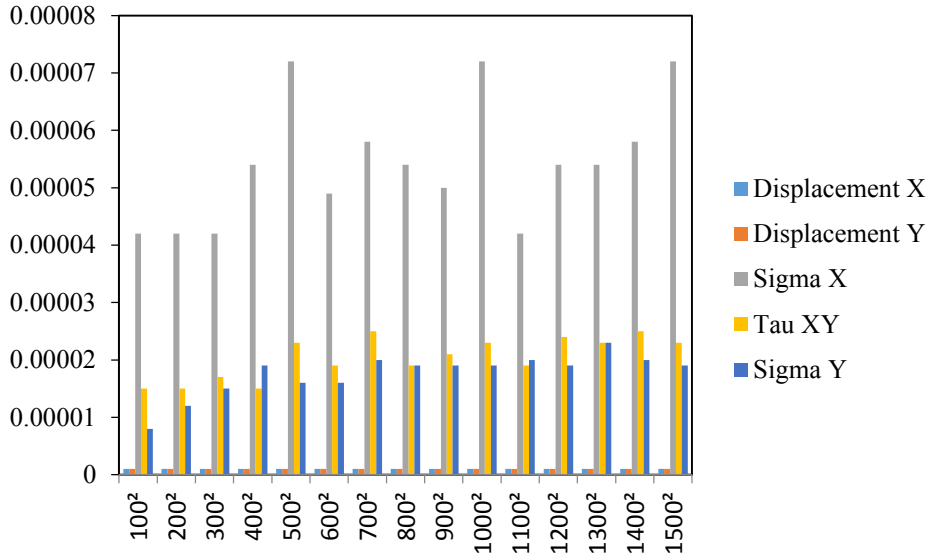


Figure 4.2: Bar chart of L_∞ Norms reflecting differences in solution vectors for various model sizes.

Table 4.3: L_∞ Norms of difference between serial solution and parallel solution.

Single precision	L_∞ Norm				
Model size	Displacement $_X$	Displacement $_Y$	σ_X	τ_{XY}	σ_Y
100²	0.000001	1E-06	4.2E-05	1.5E-05	8E-06
200²	0.000001	1E-06	4.2E-05	1.5E-05	1.2E-05
300²	1E-06	1E-06	4.2E-05	1.7E-05	1.5E-05
400²	1E-06	1E-06	5.4E-05	1.5E-05	1.9E-05
500²	1E-06	1E-06	7.2E-05	2.3E-05	1.6E-05
600²	1E-06	1E-06	4.9E-05	1.9E-05	1.6E-05
700²	1E-06	1E-06	5.8E-05	2.5E-05	2E-05
800²	1E-06	1E-06	5.4E-05	1.9E-05	1.9E-05
900²	1E-06	1E-06	5E-05	2.1E-05	1.9E-05
1000²	1E-06	1E-06	7.2E-05	2.3E-05	1.9E-05
1100²	1E-06	1E-06	4.2E-05	1.9E-05	2E-05
1200²	1E-06	1E-06	5.4E-05	2.4E-05	1.9E-05
1300²	1E-06	1E-06	5.4E-05	2.3E-05	2.3E-05
1400²	1E-06	1E-06	5.8E-05	2.5E-05	2E-05
1500²	1E-06	1E-06	7.2E-05	2.3E-05	1.9E-05

4.2 Performance Analysis

The performance of the accelerated parallel program running on a GPU is the most essential concept in this thesis. Theoretically, the parallel program based on GPU will have a much greater performance than the traditional serial program. In other words, parallel program will run faster than serial program when computing same number of exterior points for the same test case. Since it is verified in the previous section that the accuracy of the accelerated parallel program is as credible as the traditional serial program, comparison of the performance between these two programs will be carried out in this section.

The performance of a program can be measured in an easy way, which is through its execution speed (the time it takes to compute the results). In order to verify that the parallel program run faster than serial program, the circular excavation test case was taken as an example. In this case, boundary conditions (for example: traction component or displacement component) of boundary points are known in advance. In order to obtain the factor of safety of this circular excavation, we need to compute the stress components and displacement components for arbitrary points surrounding the circular excavation in the rock mass.

For the beginning of the performance comparison, the traditional BEM program was used to compute the data at field points, which the program execution time depends on the number of field points. Executing the serial program with the input file of the circular excavation case, the execution time will be recorded by an intrinsic time function. By varying the number of field points from 100^2 to 1500^2 with an increment of 100^2, an upward stretching plot of the execution time is presented, as Figure 4.3 shows. For accuracy and repeatability considerations, this data is the average of 10 executions of the program. From Figure 4.3, it is observed that the processing time of computing field points for model size 100^2 is 0.6949 seconds, which takes the least time; and the processing time for model size 1500^2 takes longest time, which is 136.9145 seconds.

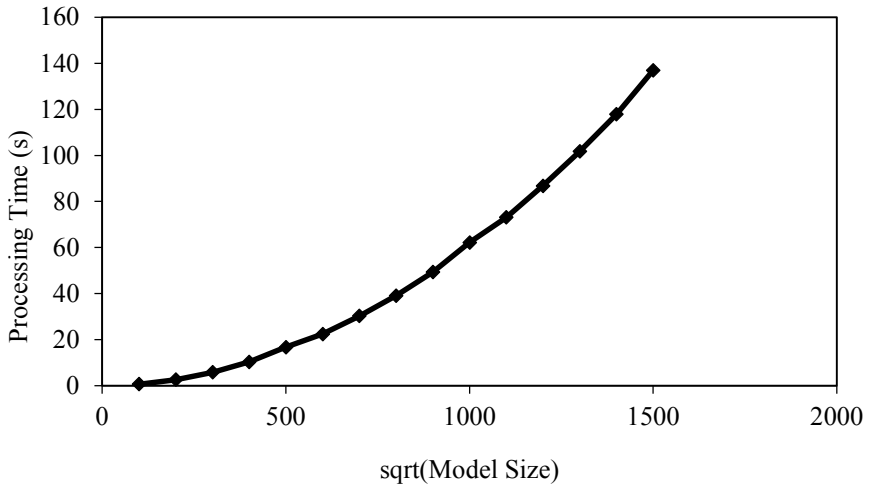


Figure 4.3: Processing time of computing field points in serial for the circular excavation case.

Then, the second step of the performance measurement was done, to execute the parallel program for the circular excavation case. Similar to the previous scenario of executing serial program, the variation of the computing field point model size was from 100^2 to 1500^2, with an increment of 100^2. For each model size, the program was executed for 10 times, as well. The average number of these 10 runs was taken and a plot of processing time versus model size was drawn, as Figure 4.4 shows. From Figure 4.4, it is seen that the processing time of GPU computing the field points for model 100^2 was 0.0037 seconds, and the processing time for model 1500^2 was 0.2378 seconds, which was the longest time in this group of tests. Comparing it to the serial processing time for the same sized models, either the 0.0037 seconds for 100^2-sized model or the 0.2378 seconds for 1500^2-sized model in GPU executions are much faster than the 0.6949 seconds for 100^2-sized model and 136.9145 seconds for 1500^2-sized model in serial executions. It is clear from this data that the GPU computing is much faster than the traditional serial computing on a CPU.

In order to present the effect of acceleration for GPU computing in an intuitive way, we can take the ratio of serial computing time over GPU computing time as the ordinate axis and the model size (square root of the number of points) as the abscissa axis to draw a plot. Figure 4.5 shows the plot, which displays the relationship between GPU speedup and the model size. From Figure 4.5, it is evident that the effect of accelerating for GPU computing has shown an upward trend with the increasing of model size. The basic speedup ratio of GPU computing over serial

computing is 187.81 when the model size is 100^2 field points. The highest speedup ratio reaches 579.43 for the field point model size of 1000^2. Moreover, a stable asymptotic effect can be achieved around 570 of the speedup ratio when the size of the field points is larger than 400^2.

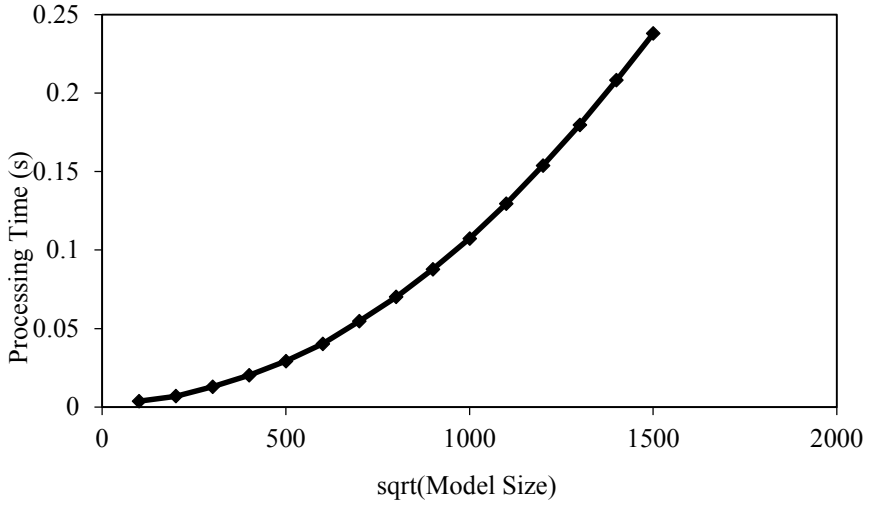


Figure 4.4: Processing time of computing field points on GPU for the circular excavation case.

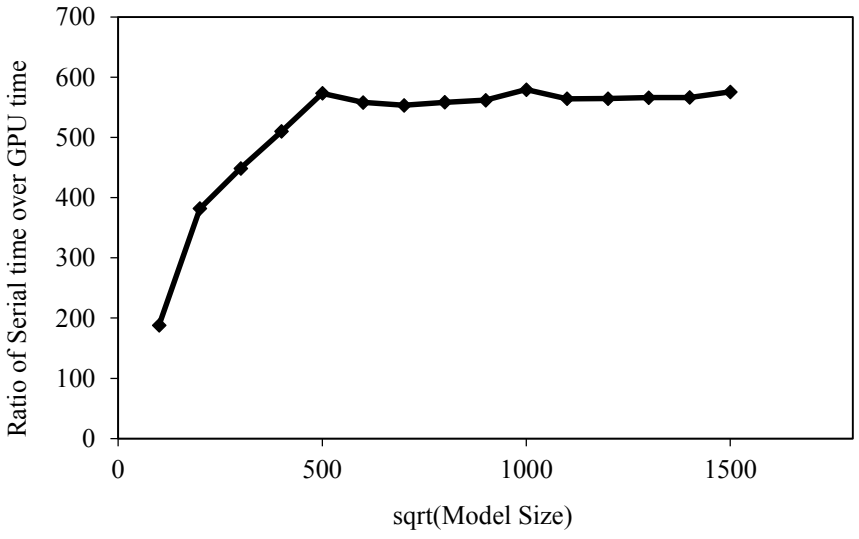


Figure 4.5: Speedup of GPU computing over serial computing for the circular excavation case.

4.3 Performance Testing Using Double-Precision Floating Point Numbers

In spite of the remarkable speedup of the GPU computing over the serial computing in the former performance test, another group of trial runs was done to take into consideration the difference between single-precision floating point numbers replaced with double-precision floating point numbers, since single-precision in numerical computing sometimes cannot provide sufficient accuracy compared to double-precision. The objective of this group of tests was to prove that the speedup of the GPU computing over the serial computing is also practical for double-precision numerical computation.

To start this group of tests, a result verification is needed. Double-precision solutions are computed from both serial program and parallel program, so that comparisons can be made between serial solutions and parallel solutions. Similar to the previous scenario where verified results in single precision, the L_∞ Norm measurement was used again to measure the difference between serial double-precision solutions and the parallel ones. The maximum absolute difference for solution vectors Displacement $_X$, Displacement $_Y$, σ_X, τ_{XY} and σ_Y were calculated respectively by using the L_∞ Norm measurement. Because of the memory limitation of the GPU computation capability, the computational model size was restricted to 800^2 field points. Therefore, the exterior point model size was varied from 100^2 to 800^2, with increments of 100^2. After calculating, the results of measurement were collected and displayed in Table 4.4, which shows the maximum absolute differences for five vectors for different model size executions. Based on Table 4.4, a bar chart was drawn to give a clearer view of the differences between double-precision serial solutions and double-precision parallel solutions. As Table 4.4 and Figure 4.6 show, solution results computed from the double-precision serial program are exactly the same (within machine precision) as the solution results from double-precision parallel program when computing Displacement $_X$ and Displacement $_Y$ for the model size smaller than 500^2; while the maximum difference was only 0.000002MPa for larger models (for the stress values), where the solution value was −55.098878MPa. These measurement results have convincingly demonstrated the accuracy of the parallel program in double-precision.

Table 4.4: L_∞ Norms of differences between double precision serial solutions and double-precision parallel solutions.

Double precision Model size	L_∞ Norm				
	Displacement $_X$	Displacement $_Y$	σ_X	τ_{XY}	σ_Y
100^2	0	0	2E−06	2E−06	2E−06
200^2	0	0	2E−06	2E−06	2E−06
300^2	0	0	2E−06	2E−06	2E−06
400^2	0	0	2E−06	2E−06	2E−06
500^2	1E−06	1E−06	2E−06	2E−06	2E−06
600^2	1E−06	1E−06	2E−06	2E−06	2E−06
700^2	1E−06	0. 000001	2E−06	2E−06	2E−06
800^2	1E−06	1E−06	2E−06	2E−06	2E−06

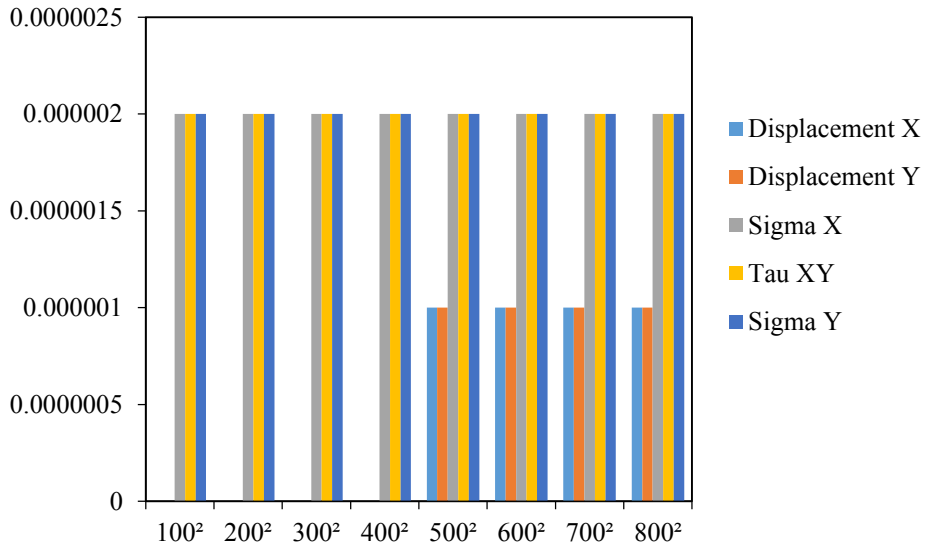


Figure 4.6: Bar chart of L_∞ Norms reflecting differences in double-precision solution vectors for various model sizes.

After verifying the accuracy of the solution results, the performance analysis can be continued to see whether the parallel program can accelerate the traditional serial program even in double-precision. The single-precision floating point input data in the serial program were transferred into double-precision floating point format. Then the input file of the circular excavation case was used to execute the serial program. By varying the model size of the exterior points, the serial program was executed. For accuracy and repeatability considerations, the program was executed for 10 times for each model size, and an average number of the execution

time was taken. The model size of the exterior points was varied from 100^2 to 1500^2, with increments of 100^2. A two-dimensional plot was drawn with the model size as the abscissa and the processing time as the ordinate, as shown in Figure 4.7. From the upward extended stretching plot, it is evident that the lowest point was at (100, 0.4296), meaning the serial program takes the shortest time (0.4296 seconds) when the exterior point number is 100^2; and the highest point was at (1500, 85.3708), meaning the processing time of serial program was 85.3708 seconds, when the exterior point number was 1500^2.

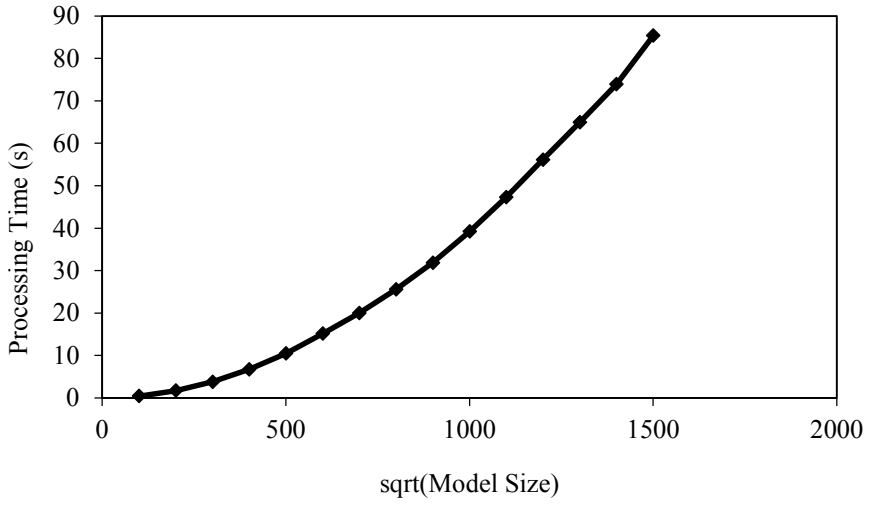


Figure 4.7: Processing time of serial program in double-precision vs. square root of model size.

The following step of this group of performance tests was to measure the speed of the parallel program. By varying the number of the exterior points, the parallel program was executed for several times and the speed of each execution was measured. Similar to serial program, the parallel program was executed 10 times of each model size for accuracy and repeatability considerations. An average execution time was taken as the final processing time for each model size. However, due to the limitations of computing capability of the current GPU device, the model size was restricted to be less than 800^2. So the varying model size was adjusted from 100^2 to 800^2, with increments of 100^2. Figure 4.8 displays an upward-stretching plot drawn from the test results, which reflects the relation between the processing time of parallel program in double-precision and the model size. It is clear in Figure 4.8 that the execution of parallel program takes the least time (0.0343 seconds) when computing 100^2 exterior points and the longest time (1.7586 seconds) when computing 800^2 exterior points.

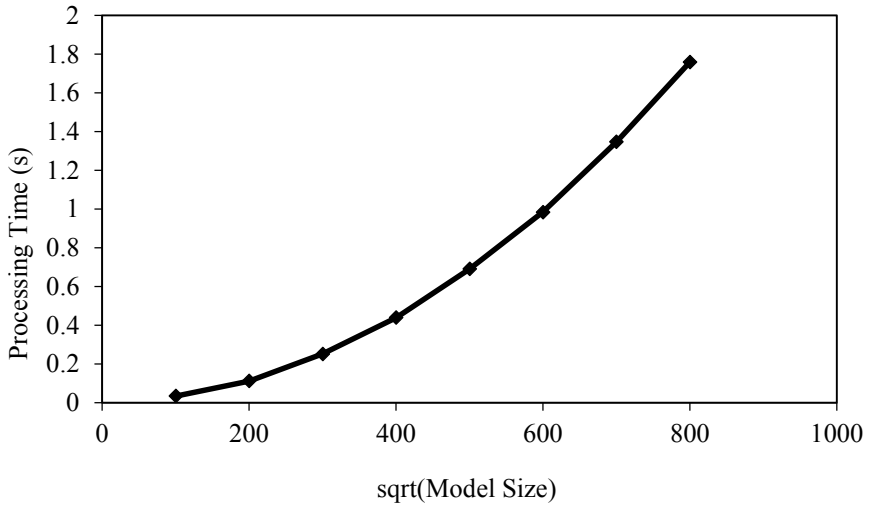


Figure 4.8: Processing time of parallel program in double-precision vs. square root of model size.

The speedup of GPU computing over serial computing in double-precision can be expressed with a plot, which is shown in Figure 4.9. The plot describes the relation between the speedup ratio of serial program processing time and the parallel program processing time and the model size. It is shown in Figure 4.9 that the lowest speedup is 12.52 when the model size is 100^2; the highest speedup is around 15 for other models, with the speedup declining for larger model sizes.

Figure 4.9: Speed up of parallel program over serial program both in double-precision for the circular excavation case.

Compared to the steady speedup of 570 in single-precision, the 15-times speedup in double-precision is kind of unimpressive at first. Though the speedup of this group of tests is not as significant as the previous one, it does prove that the parallel computing on GPU can accelerate the traditional serial computing even in double-precision. On the other hand, the accuracy verification between single-precision and double-precision shows obvious difference. From Table 4.3 and Figure 4.2, it is observed that the biggest difference in single-precision solutions is 7.2E-05. However, the biggest difference in double-precision solutions is only 0.000002, which can be found from Table 4.4 and Figure 4.6. The accuracy difference between single-precision and double-precision has precisely proved that double-precision improves accuracy in numerical computation as compared to single-precision. The reason that caused this performance result may come from the computing device. For most CPUs, the double-precision computation ability is better (more efficient) than the single-precision. On the contrary, most GPU devices show remarkable computation ability in only single-precision instead of double-precision. This is by design, since the primary purpose of a GPU is to accelerate 3D graphics, for which single-precision is sufficient, so for that purpose there is no benefit to design circuitry for efficient double-precision computation.

4.4 OpenCL Performance Optimization for Work-item Structures

In the previous sections, the feasibility of accelerating performance through parallel computing was presented and analyzed. But the acceleration performance was perhaps not the optimum with single-precision and double-precision. One question arose on whether there are other factors that may affect the acceleration performance. Therefore, a further study concerning acceleration performance evaluation was brought forward.

It is mentioned in Chapter 3, that there are two parameters in the functions where the OpenCL environment is set up, that can be investigated in further detail. These two parameters are `local_work_size` and `global_work_size`. The `global_work_size` parameter specifies the number of work-items in each dimension of the NDRange, and `local_work_size` specifies the number of work-items in each dimension of the workgroups. Organizing work-items into smaller, equally-sized workgroups leads to achieving scalability. The workgroups divides the global index space to exactly even spans and provides more coarse-grained distributions of the index space. The

advantages for work items within a workgroup are that: barrier operations to synchronize can be achieved and space address memory can be shared among work items identified with same work-group ID.

A hypothesis was build that the workgroup size can affect the acceleration performance. In order to verify this hypothesis, a number of tests were conducted by executing the single-precision circular excavation case.

The first test of trial run was for the model size of 100^2. Because the number of work-items was decided by the number of the input and output arrays in the OpenCL runtime, so that the `global_work_size` parameter was 100^2. According to the OpenCL runtime specifications, the `global_work_size` must be evenly divided by the `local_work_size`; in other words, the workgroup number must be an integer. Table 4.5 lists several applicable `local_work_size`, with their corresponding workgroup numbers.

Table 4.5: Applicable `local_work_size` and corresponding workgroup numbers for model size of 100^2.

Global work size	Local work size	Workgroup number
10000	100	100
10000	200	50
10000	250	40
10240	256	40
10000	400	25
10000	500	20
10240	512	20
10000	625	16
10000	1000	10
10240	1024	10

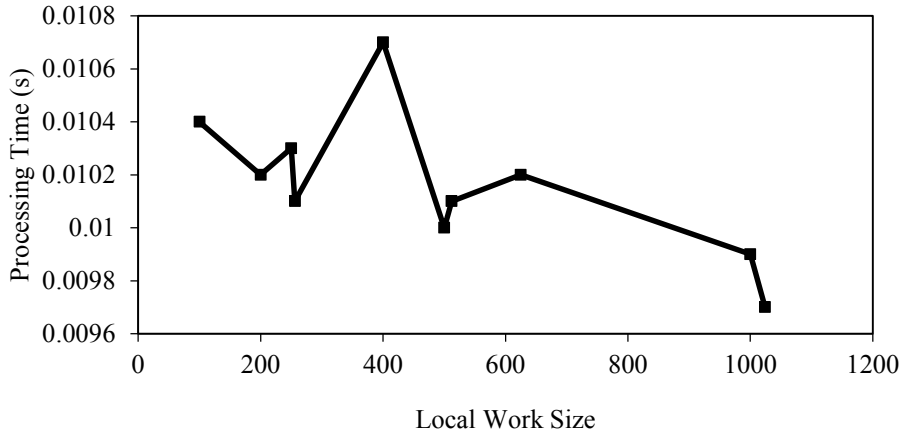


Figure 4.10: Processing time of parallel program versus local work size for model size of 100^2.

After applying the `local_work_size` into the function and executing the parallel program respectively, a number of processing times were recorded. Based on these numbers, a scatter diagram of local work size versus processing time was drawn, as Figure 4.10 shows. From the diagram, it is shown that there are fine distinctions of the processing time when the local work size are different. The program's performance was a little faster at the local work size of 256, 500, 1000 and 1024.

Same trial executions were implemented for the model sizes of 300^2, 500^2, 700^2, 1000^2 and 1500^2 as well. A set of plots reflects the local work size versus processing time for different model sizes, as shown in Figure 4.11. From the five diagrams, it can be discovered that the all the scatter diagrams have a same shape, which means that the parallel program keeps a stable performance. It is observed from the diagrams that the program runs faster at local work size 256, 512 and 1024 than others. If we sum all the processing time and keep the local work size the same, another diagram can be generated (Figure 4.12), which shows the overall processing time versus local work size. Form Figure 4.12, it is clear that the shortest processing time summation were 0.462 seconds, 0.466 seconds and 0.465 seconds when the local work size were 256, 512 and 1024, respectively.

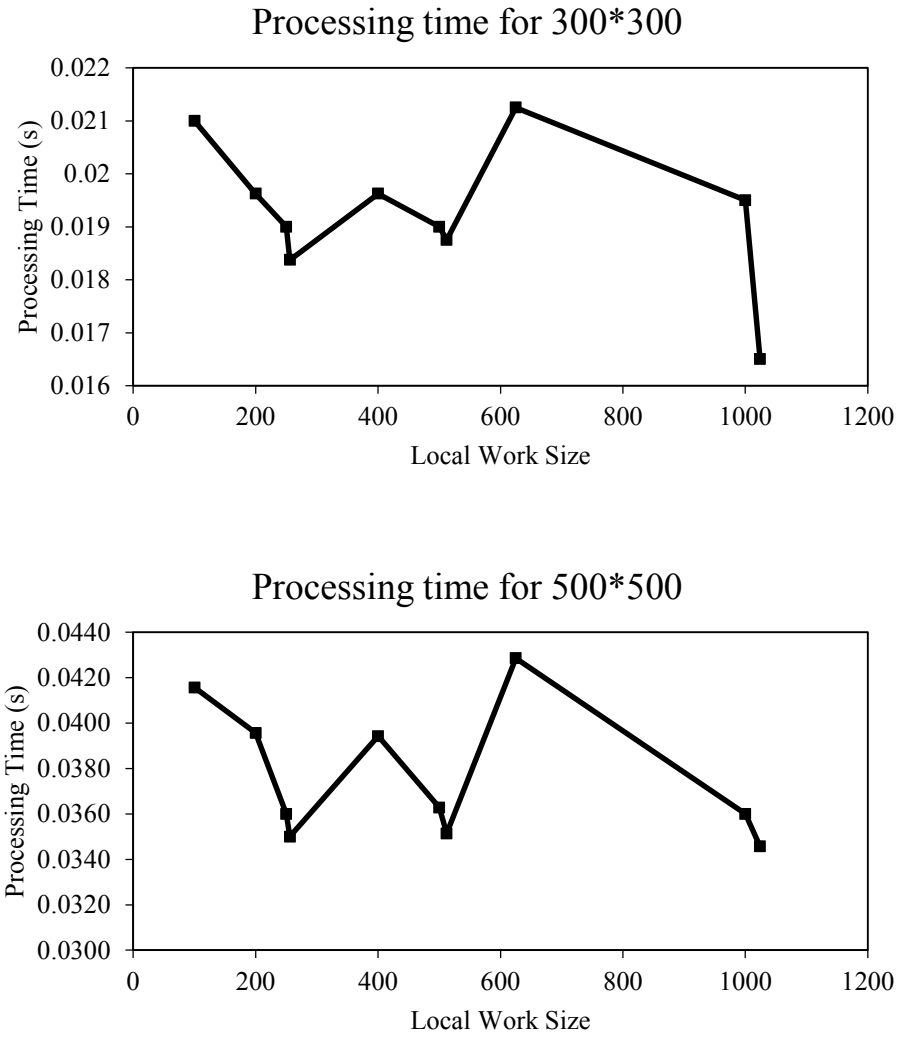
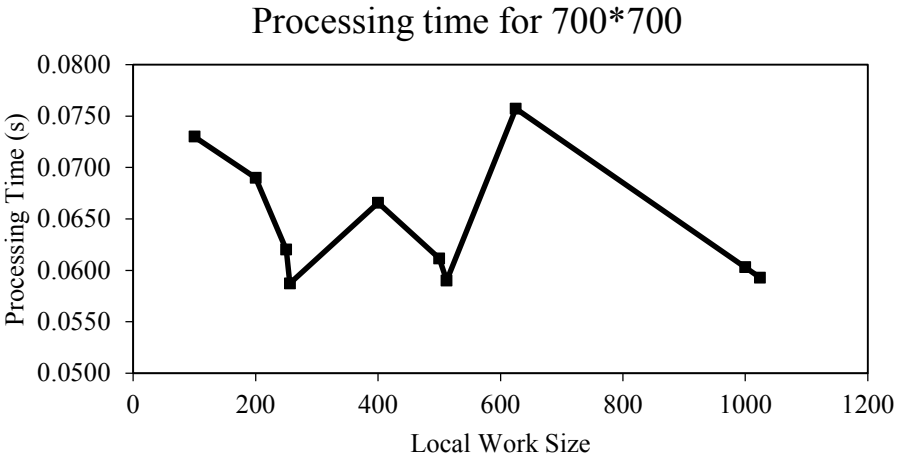
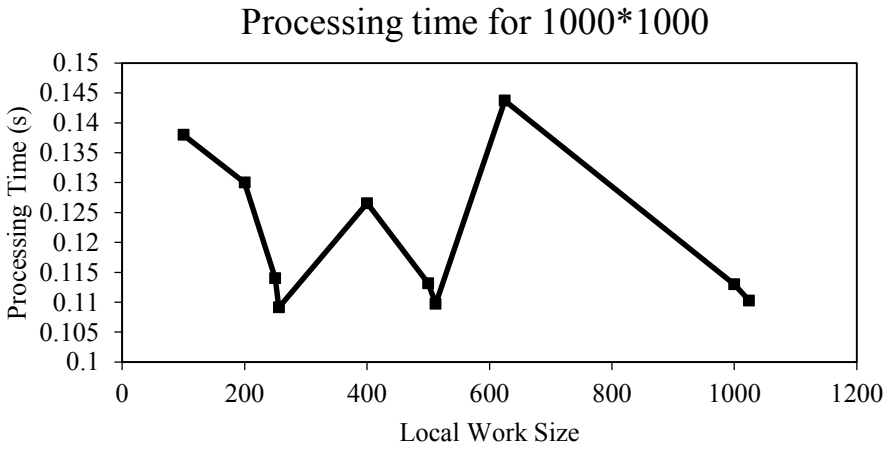


Figure 4.11: Processing time of parallel program versus local work size for different model sizes.





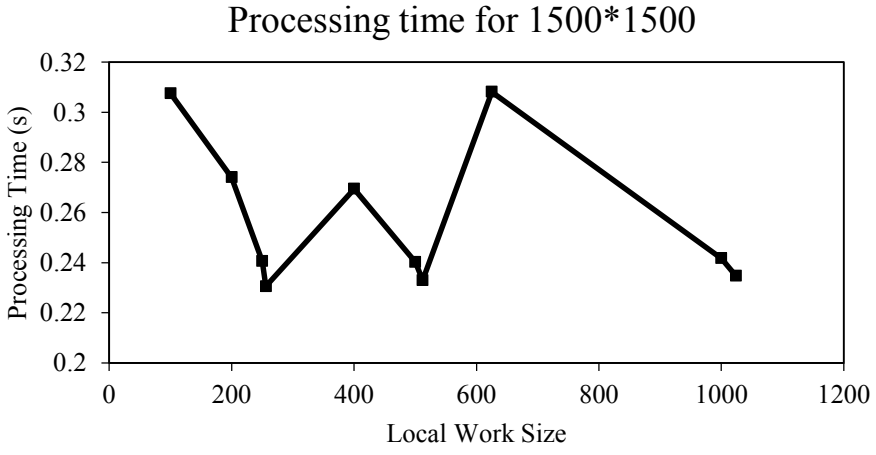


Figure 4.11 (continued): Processing time of parallel program versus local work size for different model sizes.

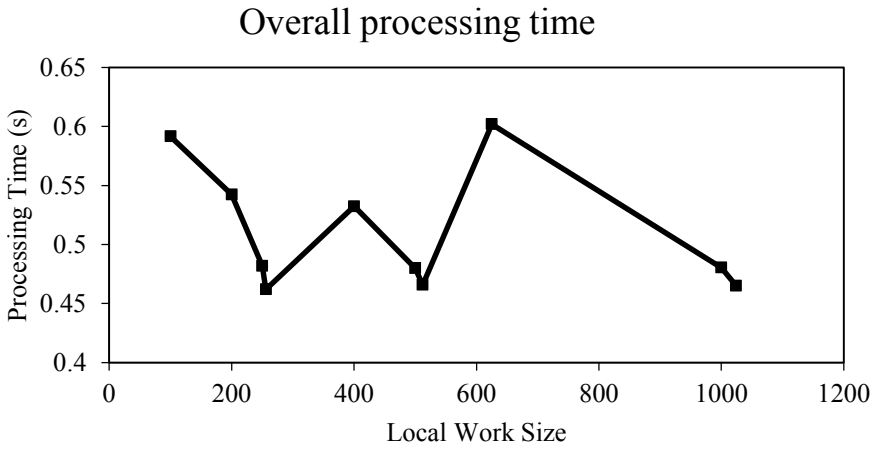


Figure 4.12: Overall processing time of parallel program versus local work size for model size of 100^2, 300^2, 500^2, 700^2, 1000^2 and 1500^2.

In conclusion, the workgroup structure does affect the acceleration performance of parallel program, though the effect is not as severe as the difference between parallel program and serial program. For a recommendation, the parameter `local_work_size` can be set to a proper number of a power of 2, such as 256, 512 and 1024, in further studies.

4.5 Conclusion

This chapter has presented a detailed execution of the BEM code for a two-dimensional circular excavation case, which was implemented for both the traditional serial program on CPU and the accelerated parallel program on GPU. Through the solution results of the circular excavation case from these two programs, the accuracy of the parallel program has been verified. At the same time, the accuracy of the double-precision implementation has been verified as well. Followed by the presentation of diagrams, which show how much speedup the parallel program can get over the serial one, and for both the single- and the double-precision formats. Later, a study of whether the workgroup structure can affect the acceleration performance has been developed, from which it was concluded that a proper `local_work_size` parameter was about 256 or 512 or 1024.

5 Application of GPU Acceleration to Practical Problems

In Chapter 4, the speedup of using parallel computing on a GPU was demonstrated for a particular example; a circular excavation. This chapter will present the implementation results of two other practical problems to see how much of a speedup they can achieve. First of all, an introduction of the two problems will be given. After that, both the traditional serial program and the GPU-accelerated parallel program will be executed for these problems respectively. Similar to the previous chapter, metrics will be used to assess the solution results and record the processing speed. Based on these results, performance analysis will be conducted on the acceleration effects.

5.1 Sample Problems

The first model problem considered (Case 1) is a two-dimensional horseshoe shaped excavation, representing a tunnel, excavated from a rock mass. As Figure 5.1 shows, the coordinates of the boundary nodes are given in the input file of this case (please see Appendix 2), with the units in meters, as well as relevant traction or displacement components of the boundary nodes. The rock mass properties used were Shear Modulus G=94500MPa, Poisson Ratio v=0.1. The displacement components and vector components of the rock mass around this excavation are needed to be computed.

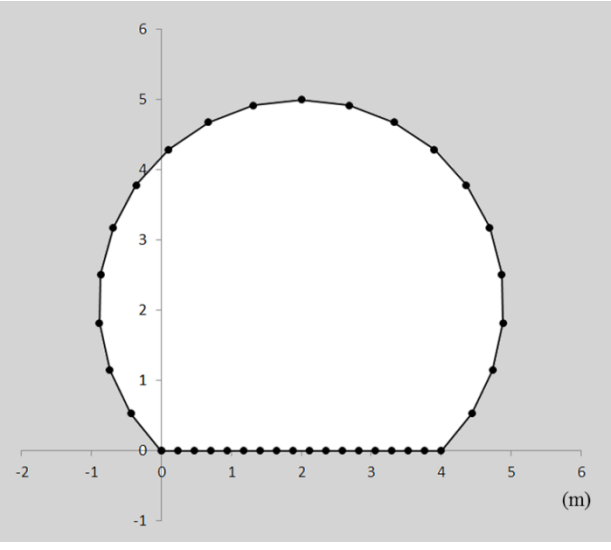


Figure 5.1: Horseshoe shaped excavation (Case 1).

Among the great majority of rock engineering works and underground excavations, a tunnel shaped as horseshoe is very common, for example, the Frejus road tunnel which links the city of Modane in France and the city of Bardonnechia in Italy was built using this horseshoe shape; the Sidi Mezghiche tunnel was constructed with a horseshoe shaped cross-section, close to the city of Sidi Mezghiche in Algeria (Panet, 1996). But before these kinds of projects being construction, a safety evaluation of stresses and displacements must be performed. This evaluation is done using the application of the numerical computations to solve the displacements and stresses in the project area. The traditional serial BEM program is one of the procedures to solve the problem. Based on the solution results worked from the serial program for the horseshoe shaped case, a layout plan of the displacements of the exterior points for model size of 100^2 and 500^2 are shown in Figure 5.2 and 5.3, respectively. From the figures, it can be found that the largest displacements are around the excavation (orange/red area). Using a finer grid, a more accurate solution results can be obtained. However, if the finer the grid size is chosen, the longer the processing time will be for the traditional serial BEM program. That's why we are seeking an acceleration procedure with the parallel computing on GPU, since efficiency values a lot in today's consulting engineering offices. From Figure 5.8, it can be found that the parallel computing processing time of Case 1 for the grid size of 500^2 was only 0.0333 seconds; while in Figure 5.7, the traditional serial computing processing time was as long as 18.0872 seconds for the same grid size.

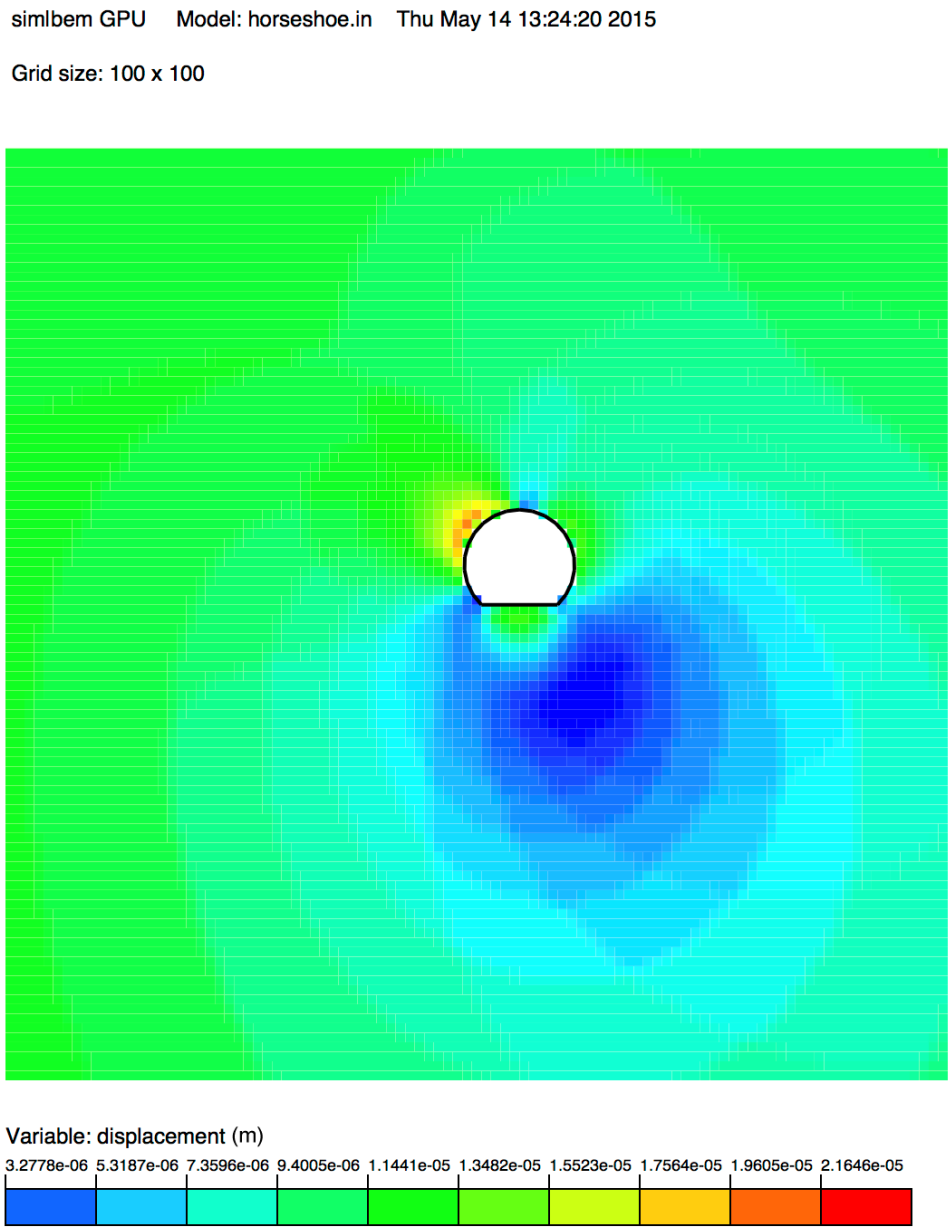


Figure 5.2: Layout plan of the displacements of exterior points for model size of 100^2.

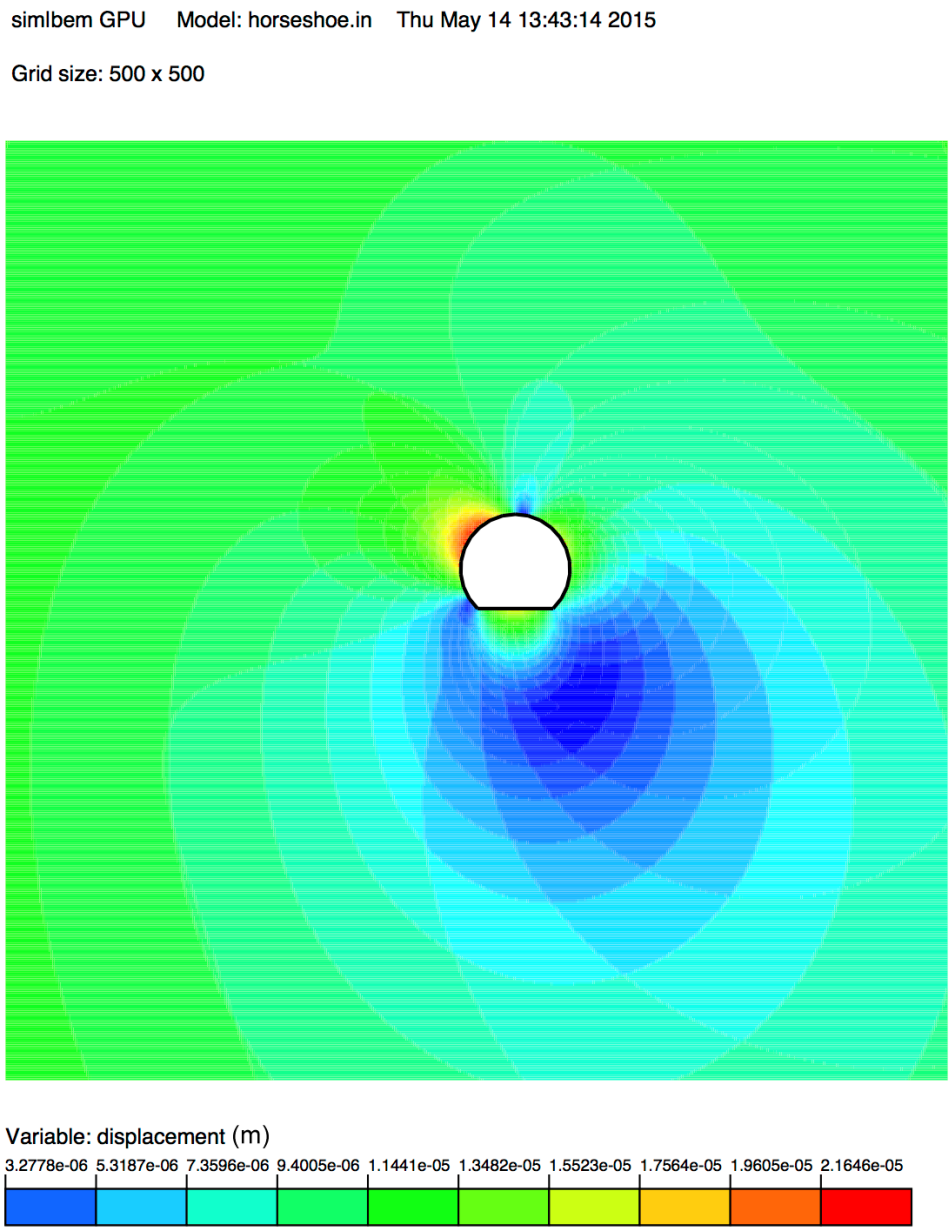


Figure 5.3: Layout plan of the displacements of exterior points for model size of 500^2.

Similarly, the second problem (Case 2) is a two-dimensional underground cavern; the cross-section of the cavern is shown in Figure 5.4. The coordinates of the boundary nodes are shown on the figure, with the units in meters. Values of the traction components or displacement

components of the boundary nodes are given in the input file of this case as well (please see Appendix 2). Other parameters of rock mass properties used were Shear Modulus G=94500MPa, Poisson Ratio v=0.1, the internal pressure was 1MPa. In order to work out the factor of safety for this cavern, we need to calculate the displacements and stresses in the rock around the cavern.

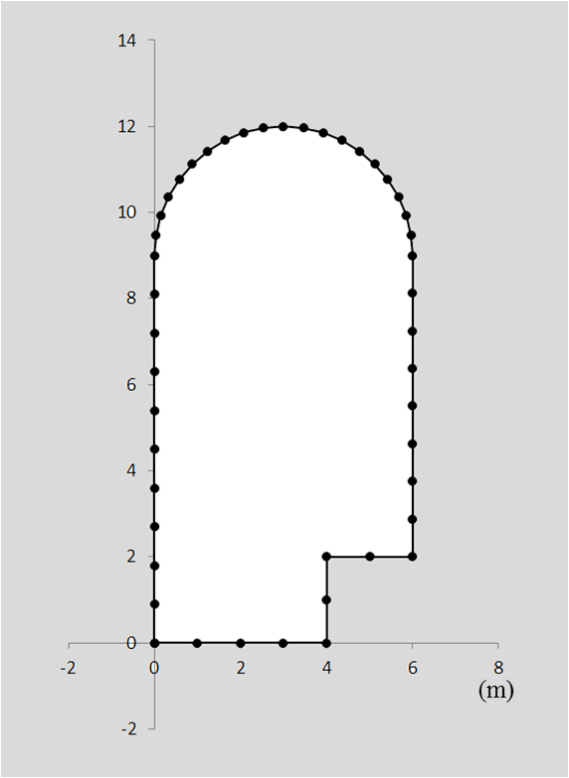


Figure 5.4: Underground cavern (Case 2).

A practical usage of Case 2 is represented by underground hydroelectric power stations. More and more underground caverns were excavated in recent years with the general considerations in advantages of underground caverns, for example: economy, safety of operation, and protection against damage and water conservation. Several studies were developed of the rock mass behaviour to check the stability of underground caverns. (Dhawan et al., 2004) used 3-D FEM to simulate underground caverns in Koyna hydroelectric project which is located in Maharashtra of India; (Franco et al., 1997) investigated the rock mass conditions of underground caverns in the Serra da Mesa Hydroelectric Power Plant which located in the Tocantins River in Brazil, with operating laboratory and in-situ tests; (Xia et al., 2007) studied the stability of an

underground power cavern constructed in Xiaolangdi Multipurpose Dam Project of China, using FEM.

In this chapter, 2-D stress analysis for underground cavern (Case 2) was conducted with the traditional serial BEM program and the accelerated parallel BEM program. From the results of these two programs, layout plans of the shear strengths (τ_{XY}) of exterior points in Case 2 were generated in Figure 5.5 and 5.6. Figure 5.5 shows the layout plans of the shear strengths for 100^2-sized exterior points and Figure 5.6 shows the layout plans of the shear strengths for 500^2-sized exterior points. It can be found in Figure 5.5 that the most overstressed points were distributed in the upper boundary of this cavern (in orange and red area), while the clearer details can be found in Figure 5.6 that almost all the points near the boundary of the caver were overstressed. The difference between the 100^2-sized model and the 500^2-sized model is so obvious, that it demonstrated the necessity in having a larger-sized model in stress analysis. Therefore, an acceleration method for executing the stress analysis with a relatively large-sized model seemed important as well.

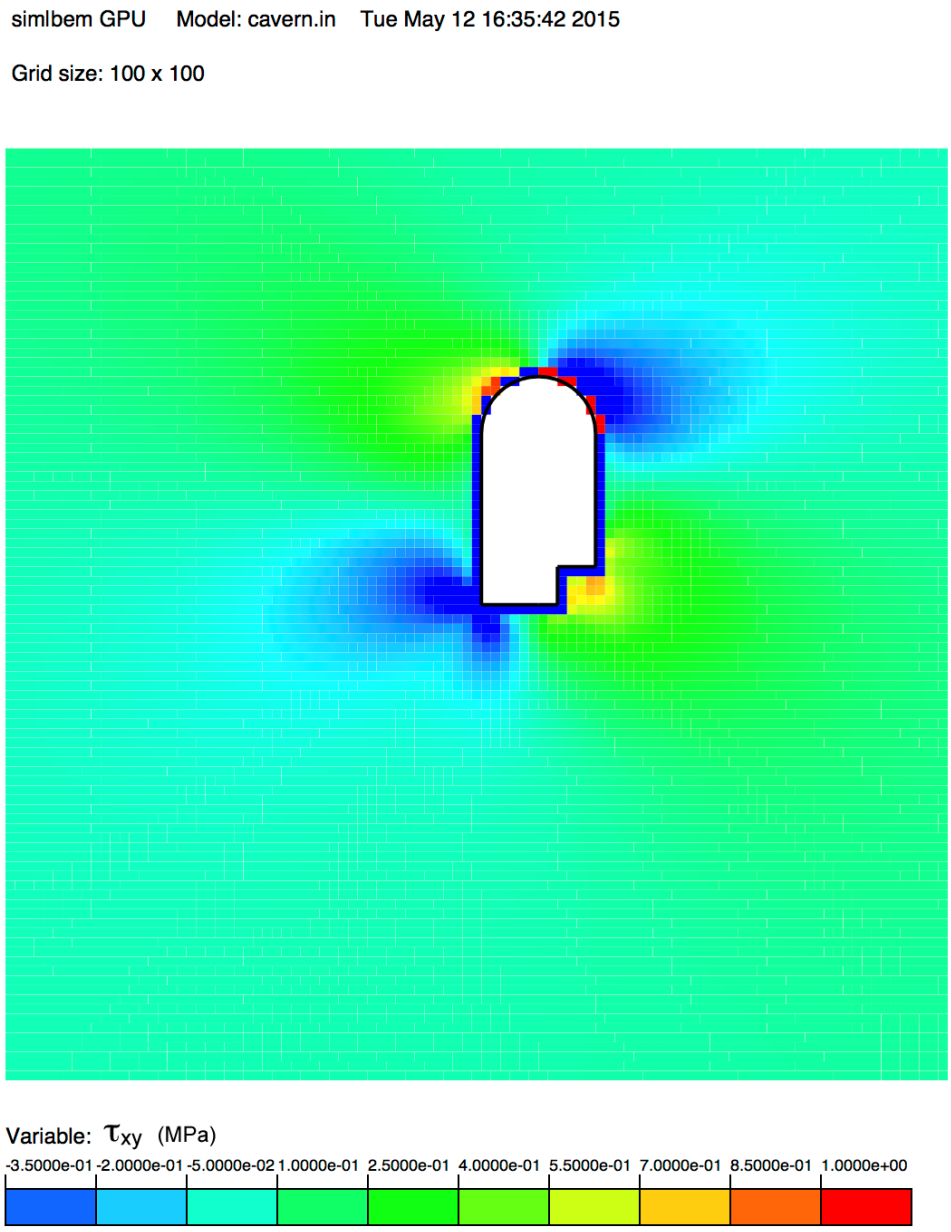


Figure 5.5: Layout plan of the shear strengths of exterior points for model size of 100^2 in Case 2.

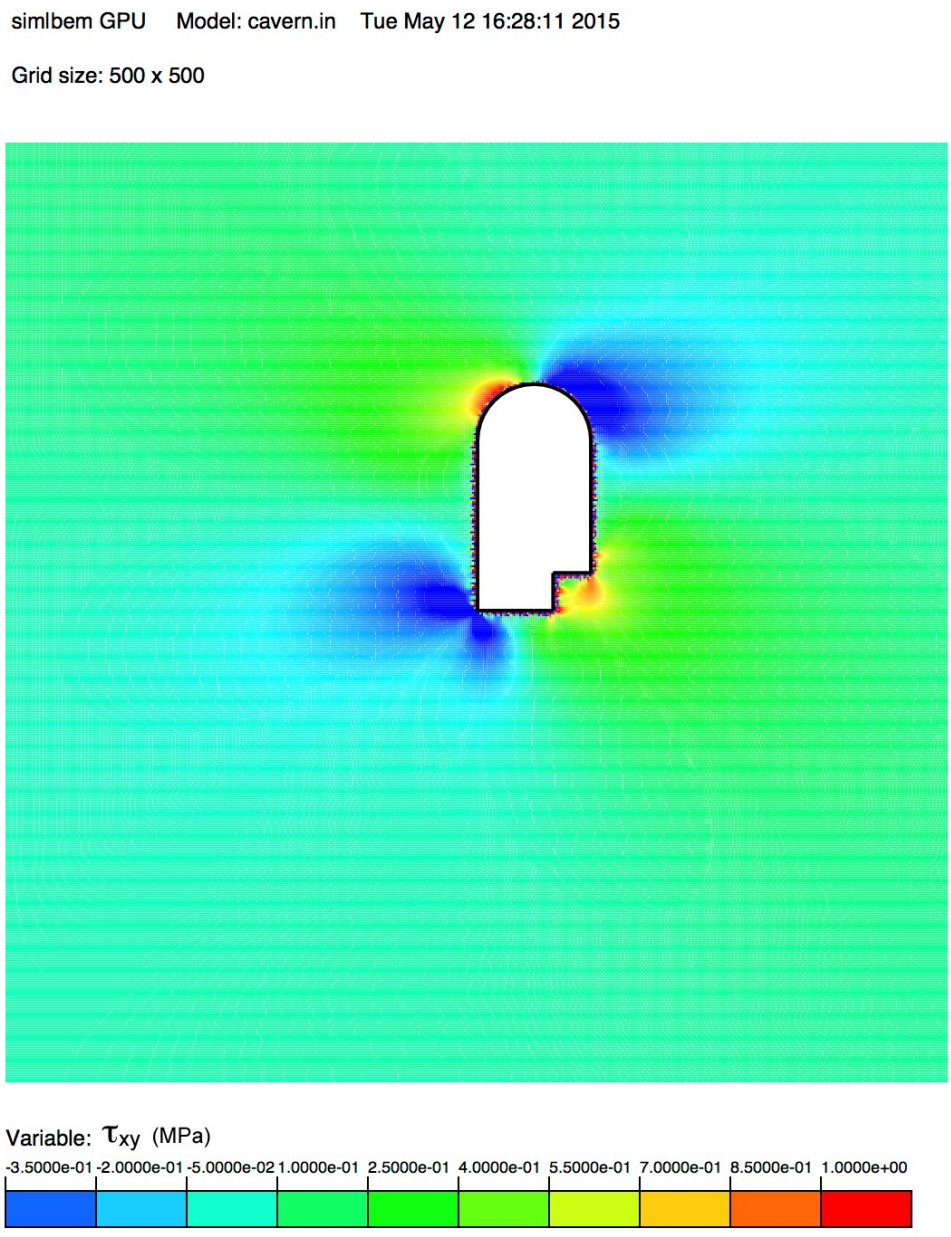


Figure 5.6: Layout plan of the shear strengths of exterior points for model size of 500^2 in Case 2.

5.2 Performance of GPU Acceleration for Practical Problems

In this section, the performance of both the serial and the parallel program will be demonstrated for Case 1 and 2. The objective of this section is to verify the acceleration effect of the parallel program over the traditional serial program; therefore all the executions will be done using single-precision floating point numbers. In addition, the parameter `local_work_size` in the parallel program is set to 1024, since it was demonstrated in Chapter 4 that such number appears to be optimum to keep the parallel program with the highest performance.

The performance analysis for Case 1 will be presented first. The traditional serial program was executed with the input file of Case 1. By varying the model size of the number of exterior points, for example, from 100^2, 200^2 … 1500^2 with the increments of 100^2, the serial program was executed for each model size, and the execution time was recorded. After ten times executions for each model size, an average number of the processing time was calculated for accuracy and repeatability considerations. Based on these values of processing time, a plot, which reflects the relation of processing time and square root of model size, can be drawn, as Figure 5.7 shows. It was observed in the plot that the relation of processing time and square root of model size has an exponential function shape. The processing time takes longer as the model size increases. That is because the processing time is connected to the size of computing grid in serial computing. The next process is to execute the parallel program for the various model sizes and to record the speed for each execution. Similar to the serial execution, the model size considered were from 100^2, 200^2 … 1500^2 with the increments of 100^2; and for each model size the program was executed ten times for accuracy and repeatability considerations. According to the average values of processing time, a line was drawn, as seen in Figure 5.8, which reflects the relation of the processing time and the square root of model size. The processing time increases with the growing model size. While the longest time for the model size 1500^2 was 0.2697 seconds, which is faster than the 156.9863 seconds for the same model size for the serial case.

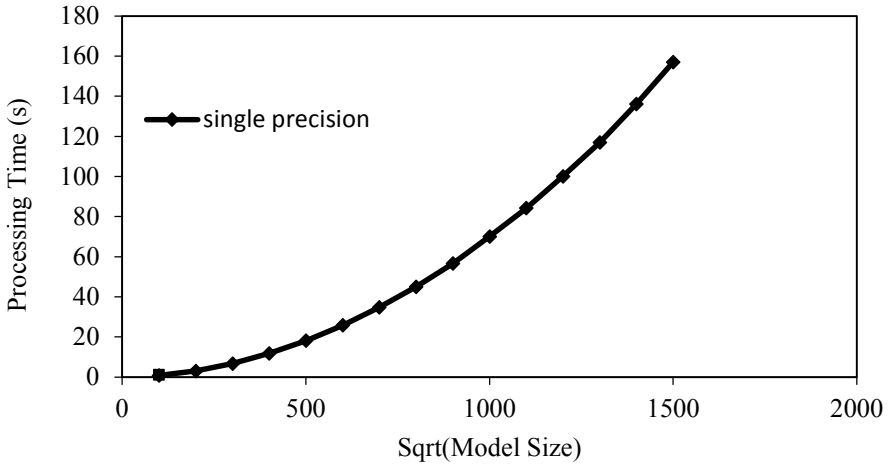


Figure 5.7: Processing time of serial computing vs. square root of model size for horseshoe shaped excavation (Case 1).

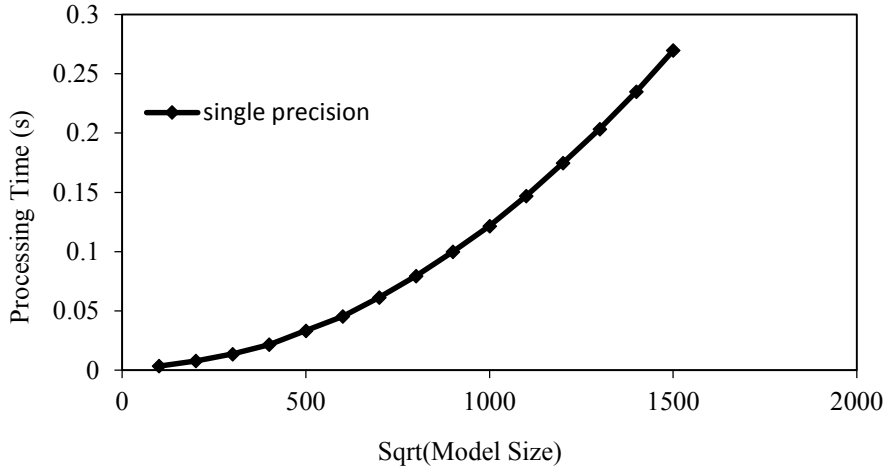


Figure 5.8: Processing time of parallel computing vs. square root of model size for horseshoe shaped excavation (Case 1).

The speedup of the parallel program over the serial program for Case 1 can be expressed as the ratio of the parallel processing time over the serial processing time. The relation of the speedup ratio to the square root of model size was drawn with a line in Figure 5.9. The shape of this line is like an exponential function, and the trend keeps rising as the model size increase. The lowest speedup ratio for Case 1 was 215.43 for model size of 100^2, while the highest speedup

ratio was 582.08 for the model size of 1500^2. A steady, asymptotic speedup was reached around 560 for the model size of 400^2 field points and larger ones.

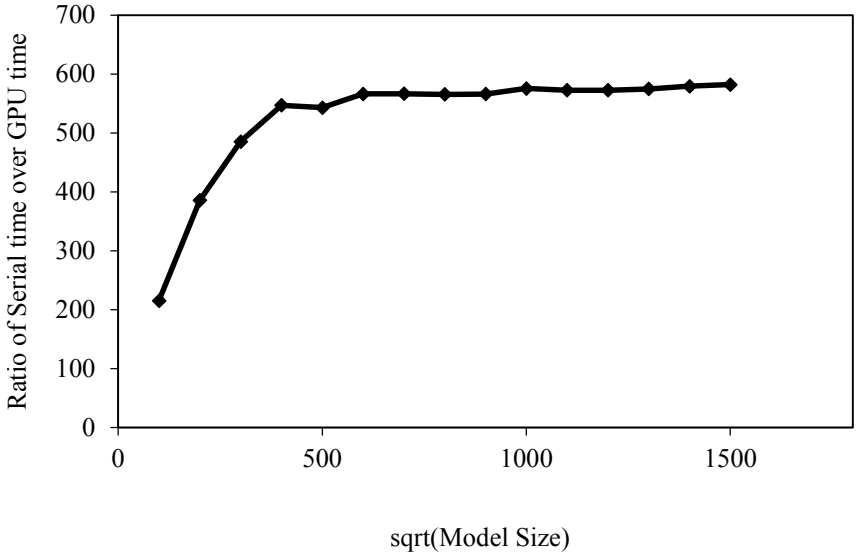


Figure 5.9: Speedup of the parallel computing over the serial computing for the horseshoe shaped excavation (Case 1).

The process of verifying the acceleration effect of parallel program over serial program for Case 2 followed the same procedure as for Case 1. The model size of computing exterior points (or field points) was varied from 100^2, 200^2 to 1500^2, with 100^2 increments. For each model size, the serial program was executed for 10 times to get a more accurate number for the processing time. From the 10 numbers of each model size, an average number was calculated as the final value of the processing time. Thus, a line was generated in Figure 5.10, which shows the variation of processing time versus square root of model size for Case 2. The processing time shows an upward trend with the increase in the model size. The shortest processing time was 0.9109 seconds when the model size was 100^2; and the longest processing time was 190.9998 seconds when the model size was 1500^2. The next step was to execute the parallel program with the input file of Case 2. The model size was varying from 100^2 to 1500^2 with every 100^2, as before. Also, the parallel program was run 10 times for each model size to count the processing times. After an average processing time was calculated for each model size, the line was drawn,

as seen in Figure 5.11. The line shows the variation of the processing time and the square root of model size. From Figure 5.11, it can be observed that the shortest processing time was 0.0034 seconds when the model size was 100^2; and the longest processing time was 0.3225 seconds when the model size was 1500^2.

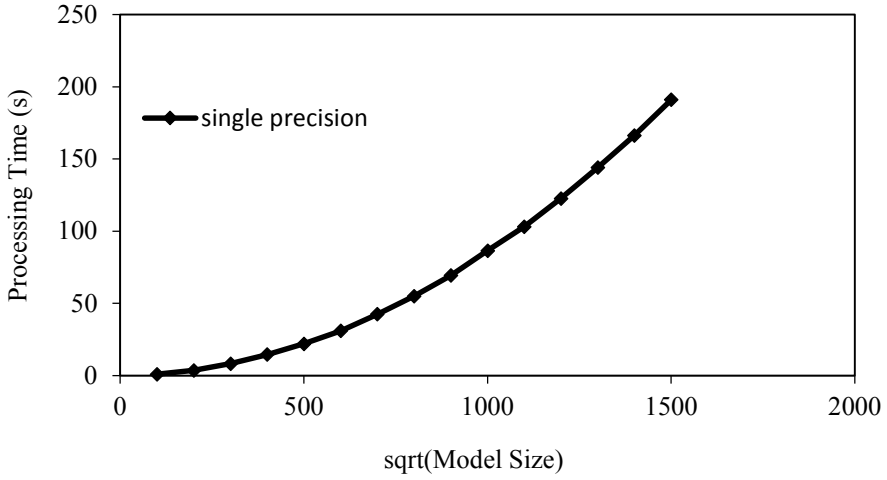


Figure 5.10: Processing time of serial computing vs. square root of model size for cavern (Case 2).

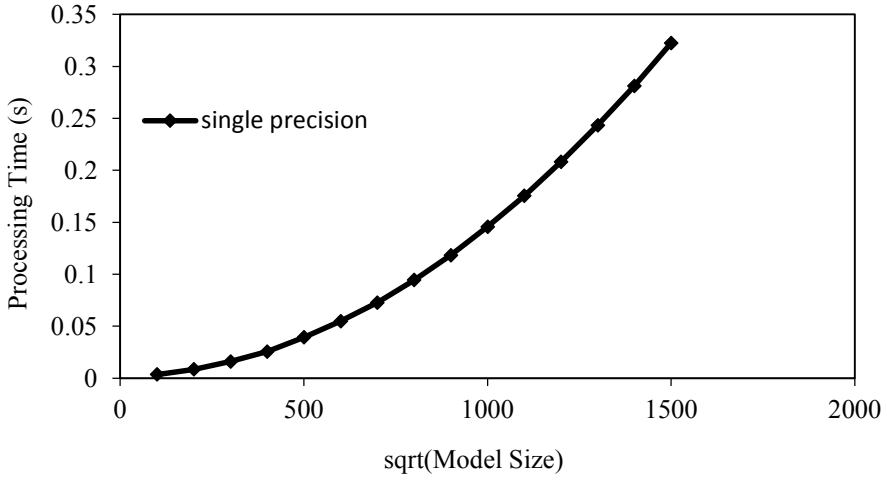


Figure 5.11: Processing time of parallel computing vs. square root of model size for cavern (Case 2).

The acceleration performance in Case 2 was counted with the ratio of the serial processing time over the parallel processing time. A plot reflects the relation of speedup and the square root

of model size is shown in Figure 5.12. The line in Figure 5.12 displays an upward trend with the increase of the model size. For the model size of 100^2, the speedup ratio was 267.91, which is the smallest acceleration performance; while the highest speedup was 594.16 when model size was 1000^2. A stable acceleration performance was achieved with the speedup ratio at around 580 for model sizes 400^2 and larger.

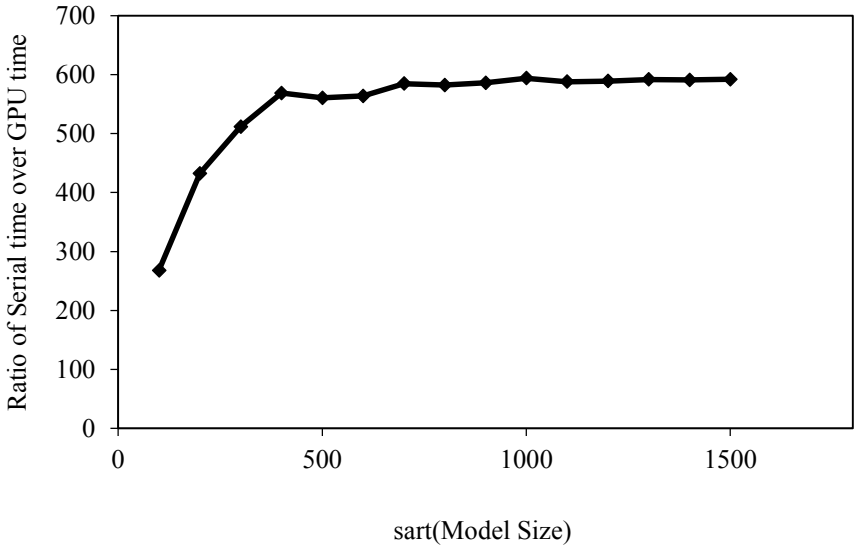


Figure 5.12: Speedup of the parallel computing over the serial computing for the cavern (Case 2).

In conclusion, the acceleration performance for the horseshoe shaped excavation case (Case 1) and the underground cavern (Case 2) was successfully demonstrated. For Case 1, the steady speedup was about 560 for model sizes 400^2 and larger; for Case 2, the steady speedup was achieved around 580 for model sizes 400^2 and larger.

6 Conclusions and Recommendations

This thesis was devoted to the study, development and analysis of an acceleration method using GPUs for computing, in parallel, the stresses and displacements around underground excavations using the BEM. An acceleration parallel algorithm was developed, which executed on a GPU using the OpenCL framework, based on the traditional serial BEM algorithm for computing field quantities. After a number of practical testing examples using the traditional serial BEM approach and the accelerated parallel BEM version, some conclusions were derived in the following section. Several recommendations are brought out as well for the future work.

6.1 Conclusions

The results from the traditional serial BEM algorithm and the accelerated parallel BEM algorithm tests can be summarized as follows:

1) Solution results from the accelerated parallel BEM algorithm have accuracy comparable to that of the serial algorithm for both single and double-precision computation. The maximum absolute difference of the solution results between the parallel program with single-precision and the serial program with single-precision was found to be quite low (7.2E-05). Similarly, the maximum absolute difference of the solution results between parallel program with double-precision and the serial program with double-precision was even smaller (2E-06). Thus there is no substantial loss of accuracy of running stress analysis on GPUs.

2) The accelerated parallel BEM algorithm has an impressive speedup over the serial implementation. For single-precision performance, the highest speedup ratio was up to 594.16 (for the cavern case). With finer computing grid size, the acceleration effect was even more pronounced. A stable speedup ratio was reached at 550 when the computing grid size was 400^2 and finer (e.g. more field points). For double-precision performance, the highest speedup ratio was 15.35; and a stable speedup ratio of around 14 was achieved.

3) On the GPUs, the size of workgroups (e.g. the number of concurrent threads) can affect the execution times. It was found that for the BEM algorithm, the values for the parameter `local_work_size` can be set to a power of 2, such as 256, 512 or 1024, for a better acceleration

performance in the accelerated parallel BEM program with the OpenCL environment. The parameter `global_work_size` in the accelerated parallel BEM program must be set as a number equal or larger than the total number of the computing grid size, according to the requirements in OpenCL documentation.

4) In order to use the parallel computing capacity of the GPU, a powerful discrete graphics card is needed, instead of some integrated graphics card, which built inside a laptop, for example. It is anticipated that dedicated OpenCL cards could achieve an even greater speedup, see item 4 in the next section.

5) The acceleration method based on the parallel computing ability of GPUs is perfectly applicable to the BEM. This is because the independence of the BEM algorithm in computing quantities at the exterior points, as demonstrated in Chapter 3.

6) The performance speed of the traditional serial BEM program is most affected by the computing grid size. The processing time of the traditional serial program shows a quadratic upward trend with the increase of computing grid size.

7) The syntax in OpenCL is similar to C++, and the programming of OpenCL functions follows a fixed pattern, which makes it simple to master for programmers.

6.2 Recommendations for Future Work

In this thesis, the acceleration method of the traditional serial BEM algorithm was developed exploiting the massively parallel computing ability of modern GPUs. With the fast advancement of the computing and programming ability of GPUs in the past two decades, more and more research interest will be attracted into the acceleration performance. Therefore, several recommendations based on the study in this thesis could be given:

1) The practical problems used in this thesis are only two-dimensional examples. More complex real-world problems for underground excavations in 3D could benefit from the acceleration of a 3D BEM program, if suitable algorithms are developed.

2) In this thesis, the performance of the accelerated parallel BEM program using double-precision floating point numbers was limited with a computing grid size of 800^2. This was

because the current NVIDIA GTX 650Ti GPU is not the most powerful device compared the newly developed ones. Perhaps further research may be focused on the performance using double-precision as more sophisticated equipment with a more powerful GPU device becomes available.

3) The platform, which executed the program on GPU was via OpenCL. Although the OpenCL is easy for programmer to handle, there are other platforms that can program the GPU, such as CUDA from NVIDIA. In a future research, comparisons can be made with the results and performance between different platforms.

4) Dedicated OpenCL cards, such as NVIDIA's Tesla GPUs could achieve an even greater speedup, which should be investigated in the future.

References

1. A (brief) history of the graphics chip: From VGA to programmable, general-purpose streaming processor. (2007). Retrieved from http://www.beyond3d.com/content/articles/29/2
2. Bacon, D. F., Cheng, P. and Shukla, S. (2012). And then there were none: A stall-free real-time garbage collector for reconfigurable hardware. *ACM SIGPLAN Notices, 47*(6) 23-34.
3. Blandford, G. E., Ingraffea, A. R. and Liggett, J. A. (1981). Two−dimensional stress intensity factor computations using the boundary element method. *International Journal for Numerical Methods in Engineering, 17*(3), 387-404.
4. Blythe, D. (2008). Rise of the graphics processor. *Proceedings of the IEEE, 96*(5), 761-778.
5. Brady, B. and Bray, J. (1978). The boundary element method for determining stresses and displacements around long openings in a triaxial stress field. *International Journal of Rock Mechanics and Mining Sciences & Geomechanics Abstracts, , 15*(1) 21-28.
6. Brown ET. (1981). Putting the NATM into perspective. *Tunnels & Tunnelling*, 13(10):13–17.
7. Buck, I., Foley, T., Horn, D., Sugerman, J., Fatahalian, K., Houston, M. and Hanrahan, P. (2004). Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics (TOG), , 23*(3) 777-786.
8. Cadzow, J. A. (1973). A finite algorithm for the minimum l_∞ solution to a system of consistent linear equations. *SIAM Journal on Numerical Analysis, 10*(4), 607-617.

9. Cayol, V., and Cornet, F. (1997). 3D mixed boundary elements for elastostatic deformation field analysis. *International Journal of Rock Mechanics and Mining Sciences, 34*(2), 275-287.
10. Crouch, S. L., Starfield, A. M., and Rizzo, F. (1983). Boundary element methods in solid mechanics. *Journal of Applied Mechanics, 50*, 704.
11. Cuda, C. (2015). Programming guide. *NVIDIA Corporation, March,*
12. Cundall, P. A., and Strack, O. D. (1979). A discrete numerical model for granular assemblies. *Geotechnique, 29*(1), 47-65.
13. Dhawan, K., Singh, D., and Gupta, I. (2004). Three-dimensional finite element analysis of underground caverns. *International Journal of Geomechanics, 4*(3), 224-228.
14. Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G., and Dongarra, J. (2012). From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing, 38*(8), 391-407.
15. Fernando, R., Haines, E., and Sweeney, T. (2004). *GPU gems: Programming techniques, tips and tricks for real-time graphics*
16. Franco, J. M., Assis, A., Mansur, W., Telles, J., and Santiago, J. (1997). Design aspects of the underground structures of the serra da mesa hydroelectric power plant. *International Journal of Rock Mechanics and Mining Sciences, 34*(3), 16. e1-16. e13.
17. Gaster, B., Howes, L., Kaeli, D. R., Mistry, P., and Schaa, D. (2012). *Heterogeneous computing with OpenCL: Revised OpenCL 1.* Newnes.
18. Harris, M. J., Baxter, W. V., Scheuermann, T., and Lastra, A. (2003). Simulation of cloud dynamics on graphics hardware. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware,* 92-101.

19. Hazegh, M., Zsaki, A. M. (2013). A framework for automatic modeling of underground excavations and optimizing three-dimensional boundary and finite element meshes derived from them—framework. *International Journal for Numerical and Analytical Methods in Geomechanics, 37*(6), 641-660.
20. Henning, J. L. (2000). SPEC CPU2000: Measuring CPU performance in the new millennium. *Computer, 33*(7), 28-35.
21. Jing, L., and Hudson, J. (2002). Numerical methods in rock mechanics. *International Journal of Rock Mechanics and Mining Sciences, 39*(4), 409-427.
22. Kaixin, L., and Lingtian, G. (2003). A review on the discrete element method. *Advances in Mechanics, 33*(4), 483-490.
23. Krüger, J., and Westermann, R. (2003). Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG), 22*(3) 908-916.
24. Kythe, P. K. (1995). *An introduction to boundary element methods* CRC press.
25. Lim, E. W. C. (2008). Master curve for the discrete-element method. *Industrial & Engineering Chemistry Research, 47*(2), 481-485.
26. Munshi, A. (2009). The OpenCL specification. *Khronos OpenCL Working Group, 1*, l1-15.
27. Nickolls, J., and Kirk, D. (2009). Graphics and computing GPUs. *Computer Organization and Design: The Hardware/Software Interface, D.A. Patterson and J.L. Hennessy, 4th Ed., Morgan Kaufmann,* , A2-A77.
28. Owens, J. D., Houston, M., Luebke, D., Green, S., Stone, J. E., and Phillips, J. C. (2008). GPU computing. *Proceedings of the IEEE, 96*(5), 879-899.

29. Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J. (2007). A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum, 26*(1) 80-113.

30. Panet, M. (1996). Two case histories of tunnels through squeezing rocks. *Rock Mechanics and Rock Engineering, 29*(3), 155-164.

31. Peiró, J., and Sherwin, S. (2005). Finite difference, finite element and finite volume methods for partial differential equations. *Handbook of materials modeling* (pp. 2415-2446) Springer.

32. Potts, D. M., and Zdravkovic, L. (1999). *Finite element analysis in geotechnical engineering: Theory*. London: Telford.

33. The world's first GPU. (1999). Retrieved from http://www.nvidia.com/page/geforce256.html

34. Xia, Y., Peng, S., Gu, Z., and Ma, J. (2007). Stability analysis of an underground power cavern in a bedded rock formation. *Tunnelling and Underground Space Technology, 22*(2), 161-165.

35. Zhao, Y., Chen, X., Sham, C., Tam, W. M., and Lau, F. C. (2011). Efficient decoding of QC-LDPC codes using GPUs. *Algorithms and architectures for parallel processing* (pp. 294-305) Springer.

36. Zsaki, A.M. (2011). GPU-accelerated stress analysis in geomechanics. *64th Canadian Geotechnical Conference and 14th Pan-American Conference on Soil Mechanics and Geotechnical Engineering*. Toronto, ON, Canada.

Appendix 1

Serial BEM Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "cbox11.h"

int main()
{
    clock_t start, end; // performance timing
    double timeSamples[10];
    /*[The lengths of arrays has been increased by one, first one is not used ]*/
    int Code[101],Dim,NN;
    float X[52],Y[52],Xm[51],Ym[51],*Xi,*Yi,Bc[101],F[101];
    float G[101][101],H[101][101],D,*stress,*displ;
    Dim=100;   /*[Dim = Max dimension of the system AX = F]*/
    int i,j; // model field point sizes
    int Lx=100;
    int Ly=100;
    L=Lx*Ly;
    size_t interiorPointsSizeT=(L+1)*sizeof(float);
    Xi=(float*)malloc(interiorPointsSizeT);
    Yi=(float*)malloc(interiorPointsSizeT);
    stress=(float*)malloc(3*interiorPointsSizeT);
    displ=(float*)malloc(2*interiorPointsSizeT);
    Input11(Xi,Yi,X,Y,Code,Bc); /*[ Read Data ]*/
    for (i=0;i<Ly;++i) {
        for (j=0;j<Lx;++j) {
           Xi[Ly*i+j+1]=4.0f+(35.0f-4.0f)*j/Lx;
           Yi[Ly*i+j+1]=0.0f+(35.0f-0.0f)*i/Ly;
        }
```

```
    }
    Sys11(X,Y,Xm,Ym,G,H,Bc,F,Code,Dim); /*[ Compute matrices G and H and form the system AX = F
]*/
    NN=2*N;    /*[Solve the system AX = F ]*/
    Solve(G,F,&D,NN,Dim);

    for (int ii=0;ii<1;ii++) {
        start=clock();
        Inter11GPU(Bc,F,Code,Xi,Yi,X,Y,stress,displ); /*[ Compute stress and displacement at
interior points ]*/
        end=clock();
        timeSamples[0]=(double)(end-start)/CLOCKS_PER_SEC;
        printPerfStats(timeSamples);
    }
    Out11(Xm,Ym,Bc,F,Xi,Yi,stress,displ); /*[Output solution at boundary nodes and interior
Points ]*/
    free(Xi);
    free(Yi);
    free(stress);
    free(displ);
    return (0);
}

void Input11(float* Xi, float* Yi, float X[52], float Y[52], int Code[101], float Bc[101])
{
    int i,lnsize,Lo; //NN,Dim,k,j
    FILE *infile,*outfile;
    char line1[100]; // Title[80],
    lnsize=120;
    printf("\nEnter the name of the input file:");
    scanf("%12s", inname);
    printf("\nEnter the name of the output file:");
    scanf("%12s", outname);
    infile=fopen(inname,"r");
    outfile=fopen(outname,"w");
    fgets(line1,lnsize,infile);
```

```
    fprintf(outfile,"%s \n",line1);
    fprintf(outfile,"Input \n");
    fscanf(infile,"%d %d %d",&N,&Lo,&M);
    fprintf(outfile,"\nNumber of Boundary Elements = %2d \n",N);
    fprintf(outfile,"Number of Interior Points = %2d \n",L);
    for (i=1;i<=5;i++) {
        fscanf(infile,"%d",&Last[i]);
    }
    fscanf(infile,"%f %f",&mu,&nu);
    fprintf(outfile, "\nShear Modulus = %10.4f \n",mu);
    fprintf(outfile,"Poisson Ratio = %10.4f \n",nu);
    if (M>0) {
        fprintf(outfile,"Number of different Boundaries = %d\n\n",M);
        for (i=1;i<=M;i++) {
               fprintf(outfile,"Last node on  boundary%2d = %2d\n\n",i,Last[i]);
        }
    }
    /*[ Read coordinates of extreme points ]*/
    fprintf(outfile,"\nCOORDINATES OF EXTREME POINTS OF THE BOUNDARY ELEMENTS\n");
    fprintf(outfile,"\n\nPoint      X                  Y\n");
    for (i=1;i<=N;i++) {
        fscanf(infile,"%f %f",&X[i],&Y[i]);
        fprintf(outfile,"%2d  %10.4f \t %10.4f \n",i,X[i],Y[i]);
    }
    /*[Read boundary conditions in Bc vector. If Code[i] = 0, the Bc[] value is a prescribed
displacement; if Code[] = 1, the Bc[] value is a prescribed traction. ]*/
    fprintf(outfile,"\nBoundary Conditions\n\n");
    fprintf(outfile," Prescribed Value        Prescribed Value\n");
    fprintf(outfile,"Node   X-direction     Code   Y-direction    Code\n");
    for (i=1;i<=N;i++) {
        fscanf(infile,"%d %f %d %f",&Code[2*i-1],&Bc[2*i-1],&Code[2*i],&Bc[2*i]);
        fprintf(outfile,"%2d \t %10.4f \t\t %2d \t %10.4f \t\t %2d\n",i, Bc[2*i-1],Code[2*i-
1],Bc[2*i],Code[2*i]);
    }
    for (i=1;i<=80;i++) {
        fprintf(outfile,"*");
```

```
    }
    fprintf(outfile,"\n");
    fclose(infile);
    fclose(outfile);
}

void Inter11GPU(float Bc[101], float F[101], int Code[101], float* Xi, float* Yi, float X[52],
float Y[52], float* stress, float* displ)
{
    int NN,i,j,k,kk,found;
    float temp,dx11,dy11,dx12,dy12,dx22,dy22,sx11,sy11,sx12,sy12,sx22,sy22;
    float H11,H12,H21,H22,G11,G12,G22;
    found=0;
    // [ Enter all displacements in  Bc and all tractions in F ]
    NN=2*N;
    for (i=1;i<=NN;i++){
        if (Code[i]>0) {
               temp=Bc[i];
               Bc[i]=F[i];
               F[i]=temp;
        }
        else {
               F[i]*=mu;
        }
    }
    // [ Compute stress and displacement at interior points]
    if (L) {
        for (k=1;k<=L;k++) {
            displ[2*k-1]=0.0f;
            displ[2*k]=0.0f;
            stress[3*k-2]=0.0f;
            stress[3*k-1]=0.0f;
            stress[3*k]=0.0f;
            for (j=1;j<=N;j++) {
                  // assumption is that M=1, e.g. there is only one excavation
                  kk=j+1;
```

```
                Quad11(Xi[k],Yi[k],X[j],Y[j],X[kk],Y[kk],&H11,&H12,&H21,&H22,&G11,&G12,&G22);
                displ[2*k-1]+=F[2*j-1]*G11+F[2*j]*G12-Bc[2*j-1]*H11-Bc[2*j]*H12;
                displ[2*k]+=F[2*j-1]*G12+F[2*j]*G22-Bc[2*j-1]*H21-Bc[2*j]*H22;
                Stress(Xi[k],Yi[k],X[j],Y[j],X[kk],Y[kk],&dx11,&dy11,&dx12,&dy12,&dx22,&dy22,
            &sx11,&sy11,&sx12,&sy12,&sx22,&sy22);
                stress[3*k-2]+=F[2*j-1]*dx11+F[2*j]*dy11-Bc[2*j-1]*sx11-Bc[2*j]*sy11;
                stress[3*k-1]+=F[2*j-1]*dx12+F[2*j]*dy12-Bc[2*j-1]*sx12-Bc[2*j]*sy12;
                stress[3*k]+=F[2*j-1]*dx22+F[2*j]*dy22-Bc[2*j-1]*sx22-Bc[2*j]*sy22;
            }
        }
    }
}

void Out11(float Xm[51], float Ym[51], float Bc[101], float F[101], float* Xi, float* Yi,
float* stress, float* displ)
{
    FILE *outfile;
    int i,k;
    outfile=fopen(outname,"a");
    fprintf(outfile,"\nResults:\n\n Boundary Nodes\n\n");
    fprintf(outfile," X        Y    Displ XDispl Y  Traction X  Traction Y\n");
    for (i=1;i<=N;i++) {
        fprintf(outfile,"(%10.4f, %10.4f)\t %10.4f\t %10.4f\t %10.4f\t
    %10.4f\n",Xm[i],Ym[i],Bc[2*i-1],Bc[2*i],F[2*i-1],F[2*i]);
    }
    if (L) {
        fprintf(outfile,"\nInterior point displacements\n\n");
        fprintf(outfile,"  Xi   Yi   Displacement X  Displacement Y\n");
        for (k=1;k<=L;k++) {
            fprintf(outfile,"(%12.6f,%12.6f) \t %12.6f \t %12.6f\n",Xi[k],Yi[k],displ[2*k-
         1],displ[2*k]);
        }
        fprintf(outfile,"\nInterior point stresses\n\n");
        fprintf(outfile,"   Xi     Yi   Sigma X    Tau XY   Sigma Y\n");
        for (k=1;k<=L;k++) {
            fprintf(outfile,"(%12.6f,%12.6f)\t %12.6f \t  %12.6f \t%12.6f\n", Xi[k],Yi[k],
```

```
         stress[3*k-2],stress[3*k-1],stress[3*k]);
        }
    }
    fclose(outfile);
}

void Quad11(float Xp, float Yp, float X1, float Y1, float X2, float Y2, float* H11, float* H12,
float* H21, float* H22, float* G11, float* G12, float* G22)
{
    float Ax,Ay,Bx,By,nx,ny,sgn,Denom,Ra,rx,ry,slope,Perp;
    float Z[]={0.0f,0.86113631f,-0.86113631f,0.33998104f,-0.33998104f};
    float W[]={0.0f,0.34785485f,0.34785485f,0.65214515f,0.65214515f};
    float Xg[5],Yg[5],HL;
    int i;
    Ax=(X2-X1)*0.5f; // /2.0;
    Bx=(X2+X1)*0.5f; // /2.0;
    Ay=(Y2-Y1)*0.5f; // /2.0;
    By=(Y2+Y1)*0.5f; // /2.0;
    nx=(Y2-Y1)/(2.0f*sqrtf(Ax*Ax+Ay*Ay));
    ny=(X1-X2)/(2.0f*sqrtf(Ax*Ax+Ay*Ay));
    if (Ax) {
        slope=Ay/Ax;
        Perp=fabsf((slope*Xp-Yp+Y1-slope*X1)/sqrtf(slope*slope+1.0f));
    }
    else {
        Perp=fabsf(Xp-X1);
    }
    // [ Determine the direction of the outward normal ]
    sgn=(X1-Xp)*(Y2-Yp)-(X2-Xp)*(Y1-Yp);
    if (sgn<0.0f) {
        Perp=-Perp;
    }
    (*H11)=0.0f;
    (*H12)=0.0f;
    (*H21)=0.0f;
    (*H22)=0.0f;
```

```
    (*G11)=0.0f;
    (*G12)=0.0f;
    (*G22)=0.0f;
    // [ Compute coefficients of the matrices G and H ]
    Denom=4.0f*pi*(1.0f-nu);
    HL=sqrtf(Ax*Ax+Ay*Ay);
    for (i=1;i<=4;i++) {
        Xg[i]=Ax*Z[i]+Bx;
        Yg[i]=Ay*Z[i]+By;
        Ra=sqrtf((Xp-Xg[i])*(Xp-Xg[i])+(Yp-Yg[i])*(Yp-Yg[i]));
        rx=(Xg[i]-Xp)/Ra;
        ry=(Yg[i]-Yp)/Ra;
        (*G11)+=((3.0f-4.0f*nu)*logf(1.0f/Ra)+rx*rx)*W[i]*HL/(2.0f*Denom*mu);
        (*G12)+=rx*ry*W[i]*HL/(2.0f*Denom*mu);
        (*G22)+=((3.0f-4.0f*nu)*logf(1.0f/Ra)+ry*ry)*W[i]*HL/(2.0f*Denom*mu);
        (*H11)-=Perp*((1.0f-2.0f*nu)+2.0f*rx*rx)/(Ra*Ra*Denom)*W[i]*HL;
        (*H12)-=(Perp*2.0f*rx*ry/Ra+(1.0f-2.0f*nu)*(nx*ry-ny*rx))*W[i]*HL/(Ra*Denom);
        (*H21)-=(Perp*2.0f*rx*ry/Ra+(1.0f-2.0f*nu)*(ny*rx-nx*ry))*W[i]*HL/(Ra*Denom);
        (*H22)-=Perp*((1.0f-2.0f*nu)+2.0f*ry*ry)*W[i]*HL/(Ra*Ra*Denom);
    }
}

void Diag11(float X1, float Y1, float X2, float Y2, float* G11, float* G12, float* G22)
{
    float Ax,Ay,SR,Denom;
    Ax=(X2-X1)*0.5f; // /2;
    Ay=(Y2-Y1)*0.5f; // /2;
    SR=sqrtf(Ax*Ax+Ay*Ay);
    Denom=4.0f*pi*mu*(1.0f-nu);
    (*G11)=SR*((3.0f-4.0f*nu)*(1.0f-logf(SR))+(X2-X1)*(X2-X1)/(4*SR*SR))/Denom;
    (*G22)=SR*((3.0f-4.0f*nu)*(1.0f-logf(SR))+(Y2-Y1)*(Y2-Y1)/(4*SR*SR))/Denom;
    (*G12)=(X2-X1)*(Y2-Y1)/(4.0f*SR*Denom);
}

void Solve(float A[101][101], float B[101], float* D, int N, int Dim)
{
```

```
int N1,i,j,k,l,k1,found;
float c;
found=2; /*[ found is a flag which is used to check if any non zero coeff is found ]*/
N1=N-1;
for (k=1;k<=N1;k++) {
    k1=k+1;
    c=A[k][k];
    if ((fabs(c)-tol)<=0.0f) {
        found=0;
        for(j=k1;j<=N;j++)  { /*[Try to Interchange rows to get Nonzero ]*/
            if ((fabs(A[j][k])-tol)>0.0f) {
                for (l=k;l<=N;l++) {
                    c=A[k][l];
                    A[k][l]=A[j][l];
                    A[j][l]=c;
                }
            c=B[k];
            B[k]=B[j];
            B[j]=c;
            c=A[k][k];
            found=1; /*[ coeff is found ]*/
            break;
            }
        }
    }
    if (!found) {
        printf("Singularity in Row %d 1",k);
        (*D)=0.0f;
        return;
    }  /*[ If no coefficient is found the control is  transferred to main ]*/
    /*[ Divide row by diagonal coefficient ]*/
    c=A[k][k];
    for (j = k1;j<=N;j++) {
        A[k][j]/=c;
    }
    B[k]/=c;
```

```
        /*[ Eliminate unknown X[k] from row i ]*/
        for (i=k1;i<=N;i++) {
            c=A[i][k];
            for (j=k1;j<=N;j++) {
                A[i][j]-=c*A[k][j];
            }
            B[i]-=c*B[k];
        }
    }
    /*[ Compute the last unknown ]*/
    if ((fabs(A[N][N])-tol)>0.0f) {
        B[N]/=A[N][N];
        /*[Apply back substitution to compute the remaining unknowns  ]*/
        for (l=1;l<=N1;l++) {
            k=N-l;
            k1=k+1;
            for (j=k1;j<=N;j++) {
                B[k]-=A[k][j]*B[j];
            }
        }
        /*[Compute the value of the determinent ]*/
        (*D)=1.0f;
        for (i=1;i<=N;i++) {
            (*D)*=A[i][i];
        }
    }
    else{
        printf("Singularity in Row %d 2",k);
        (*D)=0.0f;
    }
    return;
}

void Stress(float Xp, float Yp, float X1, float Y1, float X2, float Y2, float* dx11, float*
dy11, float* dx12, float* dy12,float* dx22, float* dy22, float* sx11, float* sy11, float* sx12,
float* sy12, float* sx22, float* sy22)
```

```
{
    float Xg[5],Yg[5],Z[5],W[5];     /*[dimension increased by 1]*/
    float Ax,Bx,Ay,By,nx,ny,slope,Perp,sgn,SR,FA,AL,Denom,rx,ry,Ra;
    int i;
    Z[1]=0.86113631f;
    Z[2]=-Z[1];
    Z[3]=0.33998104f;
    Z[4]=-Z[3];
    W[1]=0.34785485f;
    W[2]=W[1];
    W[3]=0.65214515f;
    W[4]=W[3];
    Ax=(X2-X1)*0.5f; // /2.0;
    Bx=(X2+X1)*0.5f; // /2.0;
    Ay=(Y2-Y1)*0.5f; // /2.0;
    By=(Y2+Y1)*0.5f; // /2.0;
    SR=sqrt(Ax*Ax+Ay*Ay);
    nx=(Y2-Y1)/(2.0f*SR);
    ny=(X1-X2)/(2.0f*SR);
    if (Ax) {
        slope=Ay/Ax;
        Perp=fabsf((slope*Xp-Yp+Y1-slope*X1)/sqrtf(slope*slope+1.0f));
    }
    else {
        Perp=fabsf(Xp-X1);
    }
    /*[Determine the direction of the outward normal ]*/
    sgn=(X1-Xp)*(Y2-Yp)-(X2-Xp)*(Y1-Yp);
    if (sgn<0.0f) {
        Perp=-Perp;
    }
    (*dx11)=0.0f;
    (*dy11)=0.0f;
    (*dx12)=0.0f;
    (*dy12)=0.0f;
    (*dx22)=0.0f;
```

```
(*dy22)=0.0f;
(*sx11)=0.0f;
(*sy11)=0.0f;
(*sx12)=0.0f;
(*sy12)=0.0f;
(*sx22)=0.0f;
(*sy22)=0.0f;
/*[ Compute displacement and stress coefficients ]*/
FA=1.0f-4.0f*nu;
AL=1.0f-2.0f*nu;
Denom=4.0f*pi*(1.0f-nu);
for (i=1;i<=4;i++) {
     Xg[i]=Ax*Z[i]+Bx;
     Yg[i]=Ay*Z[i]+By;
     Ra=sqrt(SQ(Xp-Xg[i])+SQ(Yp-Yg[i]));
     rx=(Xg[i]-Xp)/Ra;
     ry=(Yg[i]-Yp)/Ra;
     (*dx11)+=(AL*rx+2*cube(rx))*W[i]*SR/(Denom*Ra);
     (*dy11)+=(2*SQ(rx)*ry-AL*ry)*W[i]*SR/(Denom*Ra);
     (*dx12)+=(AL*ry+2*(SQ(rx))*ry)/(Denom*Ra)*W[i]*SR;
     (*dy12)+=(AL*rx+2*rx*SQ(ry))/(Denom*Ra)*W[i]*SR;
     (*dx22)+=(2*rx*SQ(ry)-AL*rx)/(Denom*Ra)*W[i]*SR;
     (*dy22)+=(AL*ry+2*cube(ry))/(Denom*Ra)*W[i]*SR;
     (*sx11)+=(2*Perp/Ra*(AL*rx+nu*2*rx-4*cube(rx))+4*nu*nx*SQ(rx) +AL*(2*nx*SQ(rx)+2*nx)-
     FA*nx)*2*mu/(Denom*SQ(Ra))*W[i]*SR;
     (*sy11)+=(2*Perp/Ra*(AL*ry-4*SQ(rx)*ry)+4*nu*nx*rx*ry+AL*2*ny*SQ(rx)-
     FA*ny)*2*mu/(Denom*SQ(Ra))*W[i]*SR;
     (*sx12)+=(2*Perp/Ra*(nu*ry-4*SQ(rx)*ry)+2*nu*(nx*ry*rx+ny*SQ(rx))+
     AL*(2*nx*rx*ry+ny))*2*mu/(Denom*SQ(Ra))*W[i]*SR;
     (*sy12)+=(2*Perp/Ra*(nu*rx-4*rx*SQ(ry))+2*nu*(nx*SQ(ry)+ny*rx*ry)+
     AL*(2*ny*rx*ry+nx))*2*mu/(Denom*SQ(Ra))*W[i]*SR;
     (*sx22)+=(2*Perp/Ra*(AL*rx-4*rx*SQ(ry))+4*nu*ny*rx*ry+AL*2*nx*SQ(ry)-
     FA*nx)*2*mu/(Denom*SQ(Ra))*W[i]*SR;
     (*sy22)+=(2*Perp/Ra*(AL*ry+2*nu*ry-4*cube(ry))+4*nu*ny*SQ(ry)+ AL*(2*ny*SQ(ry)+2*ny)-
     FA*ny)*2*mu/(Denom*SQ(Ra))*W[i]*SR;
}
```

```
}

void Sys11(float X[52], float Y[52], float Xm[51], float Ym[51], float G[101][101], float
H[101][101], float Bc[101], float F[101], int Code[101], int Dim)
{
    float temp;
    int i,j,k,NN,kk,found;
    found = 0;
    /*[Compute coordinates of the mid-nodes  ]*/
    X[N+1]=X[1];
    Y[N+1]=Y[1];
    for (i=1;i<=N;i++) {
        Xm[i]=(X[i]+X[i+1])/2.0;
        Ym[i]=(Y[i]+Y[i+1])/2.0;
    }
    if ((M-1)>0) {
        Xm[Last[1]]=(X[Last[1]]+X[1])/2.0;
        Ym[Last[1]]=(Y[Last[1]]+Y[1])/2.0;
        for (k=2;k<=M;k++) {
            Xm[Last[k]]=(X[Last[k]]+X[Last[k-1]+1])/2.0;
            Ym[Last[k]]=(Y[Last[k]]+Y[Last[k-1]+1])/2.0;
        }
    }
    for(i=1;i<=N;i++) {
        for(j=1;j<=N;j++) {
            if((M-1)>0.0) {
                if (!(j-Last[1])) {
                    kk=1;
                }
                else {
                    found=0;
                    for (k=2;k<=M;k++) {
                        if (!(j-Last[k])) {
                            kk=Last[k-1]+1;
                            found=1;
                            break;
```

```
                    }
                }
                if(!found) {
                    kk=j+1;
                }
            }
        }
        else {
            kk=j+1;
        }
        if (i-j) {
             Quad11(Xm[i],Ym[i],X[j],Y[j],X[kk],Y[kk],&H[2*i-1][2*j-1],&H[2*i-1][2*j],
             &H[2*i][2*j-1],&H[2*i][2*j],&G[2*i-1][2*j-1],&G[2*i-1][2*j],&G[2*i][2*j]);
             G[2*i][2*j-1]=G[2*i-1][2*j];
        }
        else {
            Diag11(X[j],Y[j],X[kk],Y[kk],&G[2*i-1][2*j-1],&G[2*i-1][2*j],&G[2*i][2*j]);
            H[(2*i-1)][(2*j-1)]=0.5;
            H[(2*i)][(2*j)]=0.5;
            H[(2*i-1)][(2*j)]=0.0;
            H[(2*i)][(2*j-1)]=0.0;
            G[(2*i)][(2*j-1)]=G[(2*i-1)][(2*j)];
        }
    }
}
/*[Reorder the columns of equation and form system matrix A which is stored in G]*/
NN=2*N;
for (j=1;j<=NN;j++) {
    if (Code[j]>0) {
        for (i=1;i<=NN;i++) {
            temp=G[i][j];
            G[i][j]=-H[i][j];
            H[i][j]=-temp;
        }
    }
    else {
```

```
            for (i=1;i<=NN;i++) {
                G[i][j]*=mu;
            }
        }
    }
    /*[ Form the right-side vector F which is stored in F]*/
    for(i=1;i<=NN;i++) {
        F[i]=0.0;
        for (j=1;j<=NN;j++) {
            F[i]+=H[i][j]*Bc[j];
        }
    }
}

void printPerfStats(const double* timeSamples)
{
     printf("CPU running time to compute field quantities at interior
     points:%f\n",timeSamples[0]);
}
```

Parallel BEM Code:

Noted that the code of functions `Input`, `Sys11`, `Solve`, `Quad11` and `Diag11` is the same with the code in Serial BEM program, thus the code will not be presented here.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include "cbox11.h"
#ifdef __MACH__
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
```

```
#endif
clock_t start, end; // performance timing
double timeSamples[10];
size_t statGlobalWorkSize;
size_t statLocalWorkSize;
char statOpenCLDevice[256];


int main()
{
    int Code[101],Dim,NN;
    float X[52],Y[52],Xm[51],Ym[51],*Xi,*Yi,Bc[101],F[101];
    float G[101][101],H[101][101],D,*stress,*displ;
    Dim=100;   /*[Dim = Max dimension of the system AX = F]*/
    int i,j;
    Lx=100;
    Ly=100;
    L=Lx*Ly;
    interiorPointsSizeT=(L+1)*sizeof(float);
    Xi=(float*)malloc(interiorPointsSizeT);
    Yi=(float*)malloc(interiorPointsSizeT);
    stress=(float*)malloc(3*interiorPointsSizeT);
    displ=(float*)malloc(2*interiorPointsSizeT);
    Input11(Xi,Yi,X,Y,Code,Bc); /*[ Read Data ]*/
    // compute interior points locations
    for (i=0;i<Ly;++i) {
        for (j=0;j<Lx;++j) {
            Xi[Ly*i+j+1]=4.0f+(35.0f-4.0f)*j/Lx;
            Yi[Ly*i+j+1]=0.0f+(35.0f-0.0f)*i/Ly;
        }
    }
    L=Lx*Ly;
    for (int ij=0;ij<1;ij++) {
        start=clock();
        /*[ Compute matrices G and H and form the system AX = F  ]*/
        Sys11(X,Y,Xm,Ym,G,H,Bc,F,Code,Dim);
        end=clock();
```

```
    timeSamples[0]=(double)(end-start)/CLOCKS_PER_SEC;
    char function_name[15] = "Sys11";
    printPerfStats(timeSamples,function_name);
}
for (int ik=0;ik<1;ik++) {
    start=clock();
    /*[Solve the system AX = F ]*/
    NN=2*N;
    Solve(G,F,&D,NN,Dim);
    end=clock();
    timeSamples[0]=(double)(end-start)/CLOCKS_PER_SEC;
    char function_name[15] = "Solve";
    printPerfStats(timeSamples,function_name);
}
for (int ii=0;ii<1;ii++) {
    start=clock();
    /*[ Compute stress and displacement at interior points ]*/
    Inter11GPU(Bc,F,Code,Xi,Yi,X,Y,stress,displ);
    end=clock();
    timeSamples[0]=(double)(end-start)/CLOCKS_PER_SEC;
    char function_name[15] = "inter11GPU";
    printPerfStats(timeSamples,function_name);
}
/*[Output solution at boundary nodes and interior Points ]*/
Out11(Xm,Ym,Bc,F,Xi,Yi,stress,displ);
free(Xi);
free(Yi);
free(stress);
free(displ);
printf("final statistics:\n");
printf("number of interior points:%i\n",L);
printf("OpenCL device:%s\n",statOpenCLDevice);
printf("globalWorksize:%i\n",(int)statGlobalWorkSize);
printf("localWorksize:%i\n",(int)statLocalWorkSize);
printf("buffer CPU->GPU:%f\n",timeSamples[7]);
printf("OpenCL solution:%f\n",timeSamples[8]);
```

```
    printf("buffer GPU->CPU:%f\n",timeSamples[9]);
    printf("total interior point solution
    time:%f\n",timeSamples[7]+timeSamples[8]+timeSamples[9]);
    return (0);
}

void Inter11GPU(float Bc[101], float F[101], int Code[101], float* Xi, float* Yi, float X[52],
float Y[52], float* stress, float* displ)
{
    int NN,i,k;
    float temp;
    // [ Enter all displacements in  Bc and all tractions in F ]
    NN=2*N;
    for (i=1;i<=NN;i++) {
        if (Code[i]>0) {
            temp=Bc[i];
            Bc[i]=F[i];
            F[i]=temp;
        }
        else {
            F[i]*=mu;
        }
    }
    // initialize output (stress and displacement) arrays
    for (k=1;k<=L;k++) {
        displ[2*k-1]=0.0f;
        displ[2*k]=0.0f;
        stress[3*k-2]=0.0f;
        stress[3*k-1]=0.0f;
        stress[3*k]=0.0f;
    }
    // [ Compute stress and displacement at interior points]
    const int sizeXY=52;
    const int sizeBcF=101;
    // storage size for buffers
    size_t datasizeXY=sizeof(float)*sizeXY;
```

```
size_t datasizeBcF=sizeof(float)*sizeBcF;
size_t datastress=sizeof(float)*3*(L+1);
size_t datadispl=sizeof(float)*2*(L+1);
size_t dataXiYi=sizeof(float)*(L+1);
// OpenCL setup
cl_int status;
// OpenCL platform
cl_uint numPlatforms=0;
cl_platform_id* platforms=NULL;
status=clGetPlatformIDs(0,NULL,&numPlatforms);
platforms=(cl_platform_id*)malloc(numPlatforms*sizeof(cl_platform_id));
status=clGetPlatformIDs(numPlatforms,platforms,NULL);
// OpenCL devices
cl_uint numDevices=0;
cl_device_id* devices=NULL;
status=clGetDeviceIDs(platforms[0],CL_DEVICE_TYPE_GPU,0,NULL,&numDevices);
devices=(cl_device_id*)malloc(numDevices*sizeof(cl_device_id));
status=clGetDeviceIDs(platforms[0],CL_DEVICE_TYPE_GPU,numDevices,devices,NULL);
char buffer[256];
status=clGetDeviceInfo(devices[0],CL_DEVICE_NAME,sizeof(buffer),buffer,NULL);
printf("code running on: %s\n",buffer);
strcpy(statOpenCLDevice,buffer);
cl_context context=NULL;
context=clCreateContext(NULL,numDevices,devices,NULL,NULL,&status);
//Create command queue
cl_command_queue queue;
queue=clCreateCommandQueue(context,devices[0],0,&status);
//Create memory buffers
cl_mem d_Xi;
cl_mem d_Yi;
cl_mem d_X;
cl_mem d_Y;
cl_mem d_F;
cl_mem d_Bc;
cl_mem d_displ;
cl_mem d_stress;
```

```
// input buffers to OpenCL kernels
d_Xi=clCreateBuffer(context,CL_MEM_READ_ONLY,dataXiYi,NULL,&status);
d_Yi=clCreateBuffer(context, CL_MEM_READ_ONLY,dataXiYi,NULL,&status);
d_X=clCreateBuffer(context,CL_MEM_READ_ONLY,datasizeXY,NULL,&status);
d_Y=clCreateBuffer(context,CL_MEM_READ_ONLY,datasizeXY,NULL,&status);
d_F=clCreateBuffer(context,CL_MEM_READ_ONLY,datasizeBcF,NULL,&status);
d_Bc=clCreateBuffer(context,CL_MEM_READ_ONLY,datasizeBcF,NULL,&status);
// output from OpenCL kernels, e.g. the solution
d_displ=clCreateBuffer(context,CL_MEM_WRITE_ONLY,datadispl,NULL,&status);
d_stress=clCreateBuffer(context,CL_MEM_WRITE_ONLY,datastress,NULL,&status);
// determine how much time is needed to transfer buffers from CPU to GPU
start=clock();
// write host data to device buffers
status=clEnqueueWriteBuffer(queue,d_Xi,CL_FALSE,0,dataXiYi,Xi,0,NULL,NULL);
status=clEnqueueWriteBuffer(queue,d_Yi,CL_FALSE,0,dataXiYi,Yi,0,NULL,NULL);
status=clEnqueueWriteBuffer(queue,d_X,CL_FALSE,0,datasizeXY,X,0,NULL,NULL);
status=clEnqueueWriteBuffer(queue,d_Y,CL_FALSE,0,datasizeXY,Y,0,NULL,NULL);
status=clEnqueueWriteBuffer(queue,d_F,CL_FALSE,0,datasizeBcF,F,0,NULL,NULL);
status=clEnqueueWriteBuffer(queue,d_Bc,CL_FALSE,0,datasizeBcF,Bc,0,NULL,NULL);
status=clEnqueueWriteBuffer(queue,d_displ,CL_FALSE,0,datadispl,displ,0,NULL,NULL);
status=clEnqueueWriteBuffer(queue,d_stress,CL_FALSE,0,datastress,stress,0,NULL,NULL);
end=clock();
timeSamples[7]=(double)(end-start)/CLOCKS_PER_SEC;
const char* filename="helper.cl";
const char* source=readSource(filename);
// determine how muc time it takes to run build & run the OpenCL program on GPU
start=clock();
cl_program program;
program=clCreateProgramWithSource(context,1,&source,NULL,&status);
status=clBuildProgram(program,1,devices,NULL,NULL,NULL);
cl_kernel kernel=NULL;
kernel=clCreateKernel(program,"helper",&status);
//Set kernel arguments
status=clSetKernelArg(kernel,0,sizeof(int),&L);
status|=clSetKernelArg(kernel,1,sizeof(int),&N);
status|=clSetKernelArg(kernel,2,sizeof(float),&nu);
```

```
    status|=clSetKernelArg(kernel,3,sizeof(float),&mu);
    status|=clSetKernelArg(kernel,4,sizeof(cl_mem),&d_Xi);
    status|=clSetKernelArg(kernel,5,sizeof(cl_mem),&d_Yi);
    status|=clSetKernelArg(kernel,6,sizeof(cl_mem),&d_X);
    status|=clSetKernelArg(kernel,7,sizeof(cl_mem),&d_Y);
    status|=clSetKernelArg(kernel,8,sizeof(cl_mem),&d_F);
    status|=clSetKernelArg(kernel,9,sizeof(cl_mem),&d_Bc);
    status|=clSetKernelArg(kernel,10,sizeof(cl_mem),&d_displ);
    status|=clSetKernelArg(kernel,11,sizeof(cl_mem),&d_stress);
    //Configure the work-item structure
    size_t globalWorkSize[1];
    size_t localWorkSize[1];
    // two parameters to vary
    globalWorkSize[0]=L;
    localWorkSize[0]=1000;
    statGlobalWorkSize=globalWorkSize[0];
    statLocalWorkSize=localWorkSize[0];
    //Enqueue the kernel for execution
    status=clEnqueueNDRangeKernel(queue,kernel,1,NULL,globalWorkSize,localWorkSize,0,NULL,NULL)
;
    clFlush(queue);
    end=clock();
    timeSamples[8]=(double)(end-start)/CLOCKS_PER_SEC;
    // determine how much time is needed to transfer buffers from CPU to GPU
    start=clock();
    //Read the output buffer back to the host
    clEnqueueReadBuffer(queue,d_displ,CL_TRUE,0,datadispl,displ,0,NULL,NULL);
    clEnqueueReadBuffer(queue,d_stress,CL_TRUE,0,datastress,stress,0,NULL,NULL);
    end=clock();
    timeSamples[9]=(double)(end-start)/CLOCKS_PER_SEC;
    //Release OpenCL resources
    clReleaseMemObject(d_Xi);
    clReleaseMemObject(d_Yi);
    clReleaseMemObject(d_X);
    clReleaseMemObject(d_Y);
    clReleaseMemObject(d_F);
```

```
    clReleaseMemObject(d_Bc);
    clReleaseMemObject(d_displ);
    clReleaseMemObject(d_stress);
    clReleaseKernel(kernel);
    clReleaseProgram(program);
    clReleaseCommandQueue(queue);
    clReleaseContext(context);
}

char* readSource(const char* kernelPath)
{
    cl_int statusread;
    FILE *fp;
    char *source;
    long int size;
    int i;
    printf("\nProgram file is: %s\n", kernelPath);
    fp = fopen(kernelPath, "rb");
    if(!fp) {
        printf("Could not open kernel file\n");
        exit(-1);
    }
    statusread = fseek(fp, 0, SEEK_END);
    if(statusread != 0) {
        printf("Error seeking to end of file\n");
        exit(-1);
    }
    size = ftell(fp);
    if(size < 0) {
        printf("Error getting file position\n");
        exit(-1);
    }
    rewind(fp);
    source = (char *)malloc(size + 1);
    for (i = 0; i < size+1; i++) {
        source[i]='\0';
```

```
    }
    if(source == NULL) {
        printf("Error allocating space for the kernel source\n");
        exit(-1);
    }
    fread(source, 1, size, fp);
    source[size] = '\0';
    fclose(fp);
    return source;
    free(source);
}

void printPerfStats(const double* timeSamples,char function_name[])
{
     printf("performance timing statistics\n");
     outfile=fopen(outname,"a+");
     if(strcmp(function_name,"inter11GPU")==0) {
         fprintf(outfile,"GPU running time to compute stress and displacement at interior
         points:%f\n",timeSamples[0]);
         printf("GPU running time to compute stress and displacement at interior
         points:%f\n",timeSamples[0]);
     }
     else if(strcmp(function_name,"Sys11")==0) {
         fprintf(outfile,"CPU running time to Compute matrices G and H and form the system AX =
         F:%f\n",timeSamples[0]);
         printf("CPU running time to Compute matrices G and H and form the system AX =
         F:%f\n",timeSamples[0]);
     }
     else if(strcmp(function_name,"Solve")==0) {
         fprintf(outfile,"CPU running time to Solve the system AX = F :%f\n",timeSamples[0]);
         printf("CPU running time to Solve the system AX = F :%f\n",timeSamples[0]);
     }
     fclose(outfile);
}
```

Kernel file: helper.cl

```
//#define pi  3.1415926
# define SQ(x)  ((x)*(x))
# define cube(x)      ((x)*(x)*(x))

void Quad11_Helper(float Xp, float Yp, float X1, float Y1, float X2, float Y2, float nu,float
mu, float* H11, float* H12, float* H21, float* H22,
                   float* G11, float* G12, float* G22)
{
    float Ax,Ay,Bx,By,nx,ny,sgn,Denom,Ra,rx,ry,slope,Perp;
    float Z[]={0.0f,0.86113631f,-0.86113631f,0.33998104f,-0.33998104f};
    float W[]={0.0f,0.34785485f,0.34785485f,0.65214515f,0.65214515f};
    float Xg[5],Yg[5],HL;
    int i;
    float pi=3.1415926f;
    Ax=(X2-X1)*0.5f;
    Bx=(X2+X1)*0.5f;
    Ay=(Y2-Y1)*0.5f;
    By=(Y2+Y1)*0.5f;
    nx=(Y2-Y1)/(2.0f*sqrt(Ax*Ax+Ay*Ay));
    ny=(X1-X2)/(2.0f*sqrt(Ax*Ax+Ay*Ay));
    if (Ax) {
        slope=Ay/Ax;
        Perp=fabs((slope*Xp-Yp+Y1-slope*X1)/sqrt(slope*slope+1.0f));
    }
    else {
        Perp=fabs(Xp-X1);
    }
    // [ Determine the direction of the outward normal ]
    sgn=(X1-Xp)*(Y2-Yp)-(X2-Xp)*(Y1-Yp);
    if (sgn<0.0f) {
        Perp=-Perp;
    }
    (*H11)=0.0f;
    (*H12)=0.0f;
```

```
(*H21)=0.0f;
(*H22)=0.0f;
(*G11)=0.0f;
(*G12)=0.0f;
(*G22)=0.0f;
// [ Compute coefficients of the matrices G and H ]
Denom=4.0f*pi*(1.0f-nu);
HL=sqrt(Ax*Ax+Ay*Ay);
float dummy1=(3.0f-4.0f*nu);
float dummy2=1.0f/(2.0f*Denom*mu);
float dummy3;
float dummy4;
float dummy5=(1.0f-2.0f*nu);
float dummy6=1.0f/(2.0f*Denom*mu);
float dummy7;
float dummy8;
float dummy9;
float dummy10;
// i=1
Xg[1]=Ax*Z[1]+Bx;
Yg[1]=Ay*Z[1]+By;
Ra=sqrt((Xp-Xg[1])*(Xp-Xg[1])+(Yp-Yg[1])*(Yp-Yg[1]));
dummy3=1.0f/Ra;
dummy4=log(dummy3);
rx=(Xg[1]-Xp)*dummy3;
ry=(Yg[1]-Yp)*dummy3;
dummy7=HL/(Ra*Denom);
dummy8=1.0f/(Ra*Ra*Denom);
dummy9=rx*rx;
dummy10=ry*ry;
(*G11)+=(dummy1*dummy4+dummy9)*W[1]*HL*dummy2;
(*G12)+=rx*ry*W[1]*HL*dummy6;
(*G22)+=(dummy1*dummy4+dummy10)*W[1]*HL*dummy2;
(*H11)-=Perp*(dummy5+2.0f*dummy9)*dummy8*W[1]*HL;
(*H12)-=(Perp*2.0f*rx*ry/Ra+dummy5*(nx*ry-ny*rx))*W[1]*dummy7;
(*H21)-=(Perp*2.0f*rx*ry/Ra+dummy5*(ny*rx-nx*ry))*W[1]*dummy7;
```

```
(*H22)-=Perp*(dummy5+2.0f*dummy10)*W[1]*HL*dummy8;
// i=2
Xg[2]=Ax*Z[2]+Bx;
Yg[2]=Ay*Z[2]+By;
Ra=sqrt((Xp-Xg[2])*(Xp-Xg[2])+(Yp-Yg[2])*(Yp-Yg[2]));
dummy3=1.0f/Ra;
dummy4=log(dummy3);
rx=(Xg[2]-Xp)*dummy3;
ry=(Yg[2]-Yp)*dummy3;
dummy7=HL/(Ra*Denom);
dummy8=1.0f/(Ra*Ra*Denom);
dummy9=rx*rx;
dummy10=ry*ry;
(*G11)+=(dummy1*dummy4+rx*rx)*W[2]*HL*dummy2;
(*G12)+=rx*ry*W[2]*HL*dummy6;
(*G22)+=(dummy1*dummy4+ry*ry)*W[2]*HL*dummy2;
(*H11)-=Perp*(dummy5+2.0f*rx*rx)*dummy8*W[2]*HL;
(*H12)-=(Perp*2.0f*rx*ry/Ra+dummy5*(nx*ry-ny*rx))*W[2]*dummy7;
(*H21)-=(Perp*2.0f*rx*ry/Ra+dummy5*(ny*rx-nx*ry))*W[2]*dummy7;
(*H22)-=Perp*(dummy5+2.0f*ry*ry)*W[2]*HL*dummy8;
// i=3
Xg[3]=Ax*Z[3]+Bx;
Yg[3]=Ay*Z[3]+By;
Ra=sqrt((Xp-Xg[3])*(Xp-Xg[3])+(Yp-Yg[3])*(Yp-Yg[3]));
dummy3=1.0f/Ra;
dummy4=log(dummy3);
rx=(Xg[3]-Xp)*dummy3;
ry=(Yg[3]-Yp)*dummy3;
dummy7=HL/(Ra*Denom);
dummy8=1.0f/(Ra*Ra*Denom);
dummy9=rx*rx;
dummy10=ry*ry;
(*G11)+=(dummy1*dummy4+rx*rx)*W[3]*HL*dummy2;
(*G12)+=rx*ry*W[3]*HL*dummy6;
(*G22)+=(dummy1*dummy4+ry*ry)*W[3]*HL*dummy2;
(*H11)-=Perp*(dummy5+2.0f*rx*rx)*dummy8*W[3]*HL;
```

```
    (*H12)-=(Perp*2.0f*rx*ry/Ra+dummy5*(nx*ry-ny*rx))*W[3]*dummy7;
    (*H21)-=(Perp*2.0f*rx*ry/Ra+dummy5*(ny*rx-nx*ry))*W[3]*dummy7;
    (*H22)-=Perp*(dummy5+2.0f*ry*ry)*W[3]*HL*dummy8;
    // i=4
    Xg[4]=Ax*Z[4]+Bx;
    Yg[4]=Ay*Z[4]+By;
    Ra=sqrt((Xp-Xg[4])*(Xp-Xg[4])+(Yp-Yg[4])*(Yp-Yg[4]));
    dummy3=1.0f/Ra;
    dummy4=log(dummy3);
    rx=(Xg[4]-Xp)*dummy3;
    ry=(Yg[4]-Yp)*dummy3;
    dummy7=HL/(Ra*Denom);
    dummy8=1.0f/(Ra*Ra*Denom);
    dummy9=rx*rx;
    dummy10=ry*ry;
    (*G11)+=(dummy1*dummy4+rx*rx)*W[4]*HL*dummy2;
    (*G12)+=rx*ry*W[4]*HL*dummy6;
    (*G22)+=(dummy1*dummy4+ry*ry)*W[4]*HL*dummy2;
    (*H11)-=Perp*(dummy5+2.0f*rx*rx)*dummy8*W[4]*HL;
    (*H12)-=(Perp*2.0f*rx*ry/Ra+dummy5*(nx*ry-ny*rx))*W[4]*dummy7;
    (*H21)-=(Perp*2.0f*rx*ry/Ra+dummy5*(ny*rx-nx*ry))*W[4]*dummy7;
    (*H22)-=Perp*(dummy5+2.0f*ry*ry)*W[4]*HL*dummy8;
}

void Stress_Helper(float Xp, float Yp, float X1, float Y1, float X2, float Y2, float nu, float
mu, float* dx11, float* dy11,float* dx12, float* dy12,  float* dx22, float* dy22, float* sx11,
float* sy11, float* sx12, float* sy12, float* sx22, float* sy22)
{
     float Xg[5],Yg[5],Z[5],W[5];
     float Ax,Bx,Ay,By,nx,ny,slope,Perp,sgn,SR,FA,AL,Denom,rx,ry,Ra;
     int i;
     float pi=3.1415926f;
     Z[1]=0.86113631f;
     Z[2]=-Z[1];
     Z[3]=0.33998104f;
     Z[4]=-Z[3];
```

```
W[1]=0.34785485f;
W[2]=W[1];
W[3]=0.65214515f;
W[4]=W[3];
Ax=(X2-X1)*0.5f;
Bx=(X2+X1)*0.5f;
Ay=(Y2-Y1)*0.5f;
By=(Y2+Y1)*0.5f;
SR=sqrt(Ax*Ax+Ay*Ay);
nx=(Y2-Y1)/(2.0f*SR);
ny=(X1-X2)/(2.0f*SR);
if (Ax) {
    slope=Ay/Ax;
    Perp=fabs((slope*Xp-Yp+Y1-slope*X1)/sqrt(slope*slope+1.0f));
}
else {
    Perp=fabs(Xp-X1);
}
sgn=(X1-Xp)*(Y2-Yp)-(X2-Xp)*(Y1-Yp);
if (sgn<0.0f) {
    Perp=-Perp;
}
(*dx11)=0.0f;
(*dy11)=0.0f;
(*dx12)=0.0f;
(*dy12)=0.0f;
(*dx22)=0.0f;
(*dy22)=0.0f;
(*sx11)=0.0f;
(*sy11)=0.0f;
(*sx12)=0.0f;
(*sy12)=0.0f;
(*sx22)=0.0f;
(*sy22)=0.0f;
FA=1.0f-4.0f*nu;
AL=1.0f-2.0f*nu;
```

```
Denom=4.0f*pi*(1.0f-nu);
// i=1
Xg[1]=Ax*Z[1]+Bx;
Yg[1]=Ay*Z[1]+By;
Ra=sqrt( (Xp-Xg[1])*(Xp-Xg[1]) + (Yp-Yg[1])*(Yp-Yg[1]) );
rx=(Xg[1]-Xp)/Ra;
ry=(Yg[1]-Yp)/Ra;
(*dx11)+=(AL*rx+2.0f*(rx*rx*rx))*W[1]*SR/(Denom*Ra);
(*dy11)+=(2.0f*(rx*rx)*ry-AL*ry)*W[1]*SR/(Denom*Ra);
(*dx12)+=(AL*ry+2.0f*((rx*rx))*ry)/(Denom*Ra)*W[1]*SR;
(*dy12)+=(AL*rx+2.0f*rx*(ry*ry))/(Denom*Ra)*W[1]*SR;
(*dx22)+=(2.0f*rx*(ry*ry)-AL*rx)/(Denom*Ra)*W[1]*SR;
(*dy22)+=(AL*ry+2.0f*(ry*ry*ry))/(Denom*Ra)*W[1]*SR;
(*sx11)+=(2.0f*Perp/Ra*(AL*rx+nu*2.0f*rx-4.0f*(rx*rx*rx))+4.0f*nu*nx*(rx*rx)+
AL*(2.0f*nx*(rx*rx)+2.0f*nx)-FA*nx)*2.0f*mu/(Denom*(Ra*Ra))*W[1]*SR;
(*sy11)+=(2.0f*Perp/Ra*(AL*ry-4.0f*(rx*rx)*ry)+4.0f*nu*nx*rx*ry+ AL*2.0f*ny*(rx*rx)-
FA*ny)*2.0f*mu/(Denom*(Ra*Ra))*W[1]*SR;
(*sx12)+=(2.0f*Perp/Ra*(nu*ry-4.0f*(rx*rx)*ry)+2.0f*nu*(nx*ry*rx+ny*(rx*rx))+
AL*(2.0f*nx*rx*ry+ny))*2.0f*mu/(Denom*(Ra*Ra))*W[1]*SR;
(*sy12)+=(2.0f*Perp/Ra*(nu*rx-4.0f*rx*(ry*ry))+2.0f*nu*(nx*(ry*ry)+ny*rx*ry)+
AL*(2.0f*ny*rx*ry+nx))*2.0f*mu/(Denom*(Ra*Ra))*W[1]*SR;
(*sx22)+=(2.0f*Perp/Ra*(AL*rx-4.0f*rx*(ry*ry))+4.0f*nu*ny*rx*ry+ AL*2.0f*nx*(ry*ry)-
FA*nx)*2.0f*mu/(Denom*(Ra*Ra))*W[1]*SR;
(*sy22)+=(2.0f*Perp/Ra*(AL*ry+2.0f*nu*ry-4.0f*(ry*ry*ry))+4.0f*nu*ny*(ry*ry)+
AL*(2.0f*ny*(ry*ry)+2.0f*ny)-FA*ny)*2.0f*mu/(Denom*(Ra*Ra))*W[1]*SR;
// i=2
Xg[2]=Ax*Z[2]+Bx;
Yg[2]=Ay*Z[2]+By;
Ra=sqrt( (Xp-Xg[2])*(Xp-Xg[2]) + (Yp-Yg[2])*(Yp-Yg[2]) );
rx=(Xg[2]-Xp)/Ra;
ry=(Yg[2]-Yp)/Ra;
(*dx11)+=(AL*rx+2.0f*(rx*rx*rx))*W[2]*SR/(Denom*Ra);
(*dy11)+=(2.0f*(rx*rx)*ry-AL*ry)*W[2]*SR/(Denom*Ra);
(*dx12)+=(AL*ry+2.0f*((rx*rx))*ry)/(Denom*Ra)*W[2]*SR;
(*dy12)+=(AL*rx+2.0f*rx*(ry*ry))/(Denom*Ra)*W[2]*SR;
(*dx22)+=(2.0f*rx*(ry*ry)-AL*rx)/(Denom*Ra)*W[2]*SR;
```

```
(*dy22)+=(AL*ry+2.0f*(ry*ry*ry))/(Denom*Ra)*W[2]*SR;
(*sx11)+=(2.0f*Perp/Ra*(AL*rx+nu*2.0f*rx-4.0f*(rx*rx*rx))+4.0f*nu*nx*(rx*rx)
+AL*(2.0f*nx*(rx*rx)+2.0f*nx)-FA*nx)*2.0f*mu/(Denom*(Ra*Ra))*W[2]*SR;
(*sy11)+=(2.0f*Perp/Ra*(AL*ry-4.0f*(rx*rx)*ry)+4.0f*nu*nx*rx*ry+AL*2.0f*ny*(rx*rx)-
FA*ny)*2.0f*mu/(Denom*(Ra*Ra))*W[2]*SR;
(*sx12)+=(2.0f*Perp/Ra*(nu*ry-4.0f*(rx*rx)*ry)+2.0f*nu*(nx*ry*rx+ny*(rx*rx))
+AL*(2.0f*nx*rx*ry+ny))*2.0f*mu/(Denom*(Ra*Ra))*W[2]*SR;
(*sy12)+=(2.0f*Perp/Ra*(nu*rx-4.0f*rx*(ry*ry))+2.0f*nu*(nx*(ry*ry)+ny*rx*ry)
+AL*(2.0f*ny*rx*ry+nx))*2.0f*mu/(Denom*(Ra*Ra))*W[2]*SR;
(*sx22)+=(2.0f*Perp/Ra*(AL*rx-4.0f*rx*(ry*ry))+4.0f*nu*ny*rx*ry+AL*2.0f*nx*(ry*ry)
-FA*nx)*2.0f*mu/(Denom*(Ra*Ra))*W[2]*SR;
(*sy22)+=(2.0f*Perp/Ra*(AL*ry+2.0f*nu*ry-4.0f*(ry*ry*ry))+4.0f*nu*ny*(ry*ry)+
AL*(2.0f*ny*(ry*ry)+2.0f*ny)-FA*ny)*2.0f*mu/(Denom*(Ra*Ra))*W[2]*SR;
// i=3
Xg[3]=Ax*Z[3]+Bx;
Yg[3]=Ay*Z[3]+By;
Ra=sqrt( (Xp-Xg[3])*(Xp-Xg[3]) + (Yp-Yg[3])*(Yp-Yg[3]) );
rx=(Xg[3]-Xp)/Ra;
ry=(Yg[3]-Yp)/Ra;
(*dx11)+=(AL*rx+2.0f*(rx*rx*rx))*W[3]*SR/(Denom*Ra);
(*dy11)+=(2.0f*(rx*rx)*ry-AL*ry)*W[3]*SR/(Denom*Ra);
(*dx12)+=(AL*ry+2.0f*((rx*rx))*ry)/(Denom*Ra)*W[3]*SR;
(*dy12)+=(AL*rx+2.0f*rx*(ry*ry))/(Denom*Ra)*W[3]*SR;
(*dx22)+=(2.0f*rx*(ry*ry)-AL*rx)/(Denom*Ra)*W[3]*SR;
(*dy22)+=(AL*ry+2.0f*(ry*ry*ry))/(Denom*Ra)*W[3]*SR;
(*sx11)+=(2.0f*Perp/Ra*(AL*rx+nu*2.0f*rx-4.0f*(rx*rx*rx))+4.0f*nu*nx*(rx*rx)
+AL*(2.0f*nx*(rx*rx)+2.0f*nx)-FA*nx)*2.0f*mu/(Denom*(Ra*Ra))*W[3]*SR;
(*sy11)+=(2.0f*Perp/Ra*(AL*ry-4.0f*(rx*rx)*ry)+4.0f*nu*nx*rx*ry+AL*2.0f*ny*(rx*rx)
-FA*ny)*2.0f*mu/(Denom*(Ra*Ra))*W[3]*SR;
(*sx12)+=(2.0f*Perp/Ra*(nu*ry-4.0f*(rx*rx)*ry)+2.0f*nu*(nx*ry*rx+ny*(rx*rx))
+AL*(2.0f*nx*rx*ry+ny))*2.0f*mu/(Denom*(Ra*Ra))*W[3]*SR;
(*sy12)+=(2.0f*Perp/Ra*(nu*rx-4.0f*rx*(ry*ry))+2.0f*nu*(nx*(ry*ry)+ny*rx*ry)
+AL*(2.0f*ny*rx*ry+nx))*2.0f*mu/(Denom*(Ra*Ra))*W[3]*SR;
(*sx22)+=(2.0f*Perp/Ra*(AL*rx-4.0f*rx*(ry*ry))+4.0f*nu*ny*rx*ry+AL*2.0f*nx*(ry*ry)
-FA*nx)*2.0f*mu/(Denom*(Ra*Ra))*W[3]*SR;
(*sy22)+=(2.0f*Perp/Ra*(AL*ry+2.0f*nu*ry-4.0f*(ry*ry*ry))+4.0f*nu*ny*(ry*ry)
```

```
    +AL*(2.0f*ny*(ry*ry)+2.0f*ny)-FA*ny)*2.0f*mu/(Denom*(Ra*Ra))*W[3]*SR;
    // i=4
    Xg[4]=Ax*Z[4]+Bx;
    Yg[4]=Ay*Z[4]+By;
    Ra=sqrt( (Xp-Xg[4])*(Xp-Xg[4]) + (Yp-Yg[4])*(Yp-Yg[4]) );
    rx=(Xg[4]-Xp)/Ra;
    ry=(Yg[4]-Yp)/Ra;
    (*dx11)+=(AL*rx+2.0f*(rx*rx*rx))*W[4]*SR/(Denom*Ra);
    (*dy11)+=(2.0f*(rx*rx)*ry-AL*ry)*W[4]*SR/(Denom*Ra);
    (*dx12)+=(AL*ry+2.0f*((rx*rx))*ry)/(Denom*Ra)*W[4]*SR;
    (*dy12)+=(AL*rx+2.0f*rx*(ry*ry))/(Denom*Ra)*W[4]*SR;
    (*dx22)+=(2.0f*rx*(ry*ry)-AL*rx)/(Denom*Ra)*W[4]*SR;
    (*dy22)+=(AL*ry+2.0f*(ry*ry*ry))/(Denom*Ra)*W[4]*SR;
    (*sx11)+=(2.0f*Perp/Ra*(AL*rx+nu*2.0f*rx-4.0f*(rx*rx*rx))+4.0f*nu*nx*(rx*rx)
    +AL*(2.0f*nx*(rx*rx)+2.0f*nx)-FA*nx)*2.0f*mu/(Denom*(Ra*Ra))*W[4]*SR;
    (*sy11)+=(2.0f*Perp/Ra*(AL*ry-4.0f*(rx*rx)*ry)+4.0f*nu*nx*rx*ry+AL*2.0f*ny*(rx*rx)
    -FA*ny)*2.0f*mu/(Denom*(Ra*Ra))*W[4]*SR;
    (*sx12)+=(2.0f*Perp/Ra*(nu*ry-4.0f*(rx*rx)*ry)+2.0f*nu*(nx*ry*rx+ny*(rx*rx))+
    AL*(2.0f*nx*rx*ry+ny))*2.0f*mu/(Denom*(Ra*Ra))*W[4]*SR;
    (*sy12)+=(2.0f*Perp/Ra*(nu*rx-4.0f*rx*(ry*ry))+2.0f*nu*(nx*(ry*ry)+ny*rx*ry)
    +AL*(2.0f*ny*rx*ry+nx))*2.0f*mu/(Denom*(Ra*Ra))*W[4]*SR;
    (*sx22)+=(2.0f*Perp/Ra*(AL*rx-4.0f*rx*(ry*ry))+4.0f*nu*ny*rx*ry+AL*2.0f*nx*(ry*ry)
    -FA*nx)*2.0f*mu/(Denom*(Ra*Ra))*W[4]*SR;
    (*sy22)+=(2.0f*Perp/Ra*(AL*ry+2.0f*nu*ry-4.0f*(ry*ry*ry))+4.0f*nu*ny*(ry*ry)
    +AL*(2.0f*ny*(ry*ry)+2.0f*ny)-FA*ny)*2.0f*mu/(Denom*(Ra*Ra))*W[4]*SR;
}

__kernel void helper(int L, int N, float nu, float mu, __global float* Xi, __global float* Yi,
__global float* X, __global float* Y, __global float* F, __global float* Bc, __global float*
displ, __global float* stress)
{
    const int idx=get_global_id(0);
    int idxPlus1;
    int j,kk;
    float dx11,dy11,dx12,dy12,dx22,dy22,sx11,sy11,sx12,sy12,sx22,sy22;
    float H11,H12,H21,H22,G11,G12,G22;
```

```
    idxPlus1=idx+1;
    if (idxPlus1<L+1) {
        for (j=1;j<=N;j++) {
            kk=j+1;
            Quad11_Helper(Xi[idxPlus1],Yi[idxPlus1],X[j],Y[j],X[kk],Y[kk],nu,mu,&H11,&H12,&H21
            ,&H22,&G11,&G12,&G22);
            displ[2*idxPlus1-1]+=F[2*j-1]*G11+F[2*j]*G12-Bc[2*j-1]*H11-Bc[2*j]*H12;
            displ[2*idxPlus1]+=F[2*j-1]*G12+F[2*j]*G22-Bc[2*j-1]*H21-Bc[2*j]*H22;
            Stress_Helper(Xi[idx+1],Yi[idx+1],X[j],Y[j],X[kk],Y[kk],nu,mu,&dx11,&dy11,&dx12,&d
            y12,&dx22,&dy22,&sx11,&sy11,&sx12,&sy12,&sx22,&sy22);
            stress[3*idxPlus1-2]+=F[2*j-1]*dx11+F[2*j]*dy11-Bc[2*j-1]*sx11-Bc[2*j]*sy11;
            stress[3*idxPlus1-1]+=F[2*j-1]*dx12+F[2*j]*dy12-Bc[2*j-1]*sx12-Bc[2*j]*sy12;
            stress[3*idxPlus1]+=F[2*j-1]*dx22+F[2*j]*dy22-Bc[2*j-1]*sx22-Bc[2*j]*sy22;
        }
    }
}
```

Appendix 2

Input file of circular excavation case:

```
32  10  1 32 0 0 0 0  94500  0.1
-0.294051 -2.98555  -0.870854 -2.87082
-1.41419 -2.64576  -1.90318 -2.31903
-2.31903 -1.90318  -2.64576 -1.41419
-2.87082 -0.870854  -2.98555 -0.294051
-2.98555  0.294051  -2.87082  0.870854
-2.64576  1.41419  -2.31903  1.90318
-1.90318  2.31903  -1.41419  2.64576
-0.870854  2.87082  -0.294051  2.98555
0.294051  2.98555  0.870854  2.87082
1.41419  2.64576  1.90318  2.31903
2.31903  1.90318  2.64576  1.41419
2.87082  0.870854  2.98555  0.294051
2.98555 -0.294051  2.87082 -0.870854
2.64576 -1.41419  2.31903 -1.90318
1.90318 -2.31903  1.41419 -2.64576
0.870854 -2.87082  0.294051 -2.98555
1  -19.509   1  -98.0785  1  -38.2683   1  -92.388
1  -55.557   1  -83.147   1  -70.7107   1  -70.7107
1  -83.147   1  -55.557   1  -92.388    1  -38.2683
1  -98.0785  1  -19.509   1  -100.      1  0.
1  -98.0785  1   19.509   1  -92.388    1  38.2683
1  -83.147   1  55.557    1  -70.7107   1  70.7107
1  -55.557   1  83.147    1  -38.2683   1   92.388
1  -19.509   1   98.0785  0  0.         1  100.
1  19.509  1  98.0785  1  38.2683  1  92.388
1  55.557  1  83.147   1  70.7107  1  70.7107
1  83.147  1  55.557   1  92.388   1  38.2683
1  98.0785  1  19.509  1  100.     0   0.
1  98.0785  1  -19.509  1  92.388  1  -38.2683
1  83.147  1  -55.557  1  70.7107  1  -70.7107
```

```
1  55.557  1  -83.147  1  38.2683  1  -92.388
1  19.509  1  -98.0785  0  0.  1  -100.
```

Input file of horse-shoe shaped excavation (Case 1):

```
37  10  1 37 0 0 0 0  94500  0.1
0.0 0.0 -0.438808773237683 0.530856358525801
-0.740057824344047 1.15021943626385 -0.886755359531386 1.82315438559223
-0.870626981725536 2.51170466573667 -0.692582403567252 3.1770329614269
-0.362664135550676 3.78161178117382 0.100518956482631 4.29134017562727
0.670841274603381 4.67746718363242 1.31613408936182 4.91821351502455
2 5 2.68386591063818 4.91821351502455
3.32915872539662 4.67746718363242 3.89948104351737 4.29134017562727
4.36266413555068 3.78161178117382 4.69258240356725 3.1770329614269
4.87062698172554 2.51170466573667 4.88675535953139 1.82315438559223
4.74005782434405 1.15021943626385 4.43880877323768 0.5308563585258
4 0 3.76470588235294 0.0
3.52941176470588 0.0 3.29411764705882 0.0
3.05882352941176 0.0 2.82352941176471 0.0
2.58823529411765 0.0 2.35294117647059 0.0
2.11764705882353 0.0 1.88235294117647 0.0
1.64705882352941 0.0 1.41176470588235 0.0
1.17647058823529 0.0 0.941176470588235 0.0
0.705882352941176 0.0 0.470588235294117 0.0
0.235294117647058 0.0
0 0.0   1 -0.637118966       1 -0.899270907   1 -0.437392084
1 -0.977053435 1  -0.212994333       1  -0.999725779  1 0.023417251
1 -0.966009117 1 0.258507998  1 -0.877805222  1 0.479017738
1 -0.740089185 1 0.672508734  1 -0.560628809  1 0.82806723
1 -0.349546461 1 0.936919032  1 -0.118748129   1 0.992924409
1 0.118748129 0 0.0   1 0.349546461  1 0.936919032
1 0.560628809  1 0.82806723   1  0.740089185  1 0.672508734
1 0.877805222  1 0.479017738  1 0.966009117   1 0.258507998
1 0.999725779  1 0.023417251  1 0.977053435   1 -0.212994333
1 0.899270907  1 -0.437392084 1  0.770765479  1 -0.637118966
```

```
0 0.0   1 -1.0 1 0.0 1 -1.0
1 0.0  1 -1.0  1 0.0  1 -1.0
1 0.0   1 -1.0 1 0.0  1 -1.0
1 0.0  1 -1.0  1 0.0  1 -1.0
1 0.0  1 -1.0  1 0.0  1 -1.0
1 0.0   1 -1.0 1 0.0  1 -1.0
1 0.0  1 -1.0  1 0.0  1 -1.0
1 0.0  1 -1.0  1 0.0  1 -1.0
1 0.0  1 -1.0
```

Input file of underground cavern (Case 2):

```
46  10  1 46 0 0 0 0  94500  0.1
0.0 0.0 0.0 0.899999999999998
0.0 1.8 0.0 2.7
0.0 3.6 0.0 4.5
0.0 5.4 0.0 6.3
0.0 7.2 0.0 8.1
0.0 9.0 0.0369349782145871 9.46930339512069
0.14683045111454 9.92705098312484 0.326980427434897 10.3619714992186
0.572949016875158 10.7633557568774 0.878679656440358 11.1213203435596
1.23664424312258 11.4270509831248 1.63802850078136 11.6730195725651
2.07294901687516 11.8531695488855 2.53069660487931 11.9630650217854
3.0 12.0 3.46930339512069 11.9630650217854
3.92705098312484 11.8531695488855 4.36197149921864 11.6730195725651
4.76335575687742 11.4270509831248 5.12132034355964 11.1213203435596
5.42705098312484 10.7633557568774 5.6730195725651 10.3619714992186
5.85316954888546 9.92705098312484 5.96306502178541 9.46930339512069
6.0 9.0 6.0 8.125
6.0 7.25 6.0 6.375
6.0 5.5 6.0 4.625
6.0 3.75 6.0 2.875
6.0 2.0 5.0 2.0
4.0 2.0 4.0 1.0
4.0 0.0 3.0 0.0
```

```
2.0 0.0 1.0 0.0
1 -1.0 1 0.0 1 -1.0 1 0.0
1 -1.0 1 0.0 1 -1.0 1 0.0
1 -1.0 1 0.0 1 -1.0 1 0.0
1 -1.0 1 0.0 1 -1.0 1 0.0
1 -1.0 1 0.0 1 -1.0 1 0.0
1 -0.996917334 1 0.078459096 1 -0.97236992 1 0.233445364
1 -0.923879533 1 0.382683432 1 -0.852640164 1 0.522498565
1 -0.760405966 1 0.649448048 1 -0.649448048 1 0.760405966
1 -0.522498565 1 0.852640164 1 -0.382683432 1 0.923879533
1 -0.233445364 1 0.97236992  1 -0.078459096 1 0.996917334
1 0.078459096 1 0.996917334 1 0.233445364 1 0.97236992
1 0.382683432 1 0.923879533  1 0.522498565 1 0.852640164
1 0.649448048 1 0.760405966 1 0.760405966 1 0.649448048
1 0.852640164 1 0.522498565 1 0.923879533 1 0.382683432
1 0.97236992 1 0.233445364 1 0.996917334 1 0.078459096
1 1.0 1 0.0 1 1.0 1 0.0
1 1.0 1 0.0 1 1.0 1 0.0
1 1.0 1 0.0 1 1.0 1 0.0
1 1.0 1 0.0 1 0.0 1 -1.0
1 0.0 1 -1.0 1 1.0 1 0.0
1 1.0 1 0.0 1 0.0 1 -1.0
1 0.0 1 -1.0 1 0.0 1 -1.0
1 0.0 1 -1.0 1 0.0 1 -1.0
```